Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN SISTEMAS DE INFORMACIÓN

Trabajo Fin de Grado

Investigación y Desarrollo de Técnicas de Scraping

Autor: Urbano José Villanueva Rodríguez

Tutor/es: Manuel Sánchez Rubio

2019

Agradecimientos

A Carmen,

por ser todo lo que tengo

y por estar siempre ahí

Índice de contenido

Preliminares

Resumen

En el desarrollo de esta obra se trata de conocer estado del arte, analizar herramientas actuales y tratar de avanzar en la investigación de la extracción de datos para diversos fines a través del web scraping o el crawling de sitios web.

Se elaboran 3 patrones básicos de búsqueda: recursivo, buscadores y e-commerce, para estandarizar diferentes tipos de búsqueda. Utilizando patrones de diseño se elabora un sistema eficaz, modular y totalmente escalable. Se implementa, además, un buscador para filtrar las direcciones según el contenido deseado.

Abstract

In the development of this work is to know state of the art, analyze current tools and try to advance in the research of data extraction for various purposes through web scraping and crawling of websites.

3 basic search patterns are elaborated: recursive, search engines and e-commerce, to standardize different types of searches. Using design patterns, an efficient, modular and fully scalable system is developed. A search engine is also implemented to filter the addresses according to the desired content.

Palabras clave - keywords

Jsoup, Java, Crawler, Scraping, Data mining.

Introducción

¿Qué es el web scraping?

Podemos definir el Web Scraping como un conjunto de técnicas que nos permiten, de forma automática, extraer datos de sitios web. En definitiva, se trata de obtener, analizar y conocer el código HTML devuelto por cualquier sitio tras una petición HTTP:GET.

A simple vista, puede parecer algo más complejo o abstracto de lo que realmente se trata pero, en definitiva, no es más que buscar palabras, tipos de datos o patrones en la codificación HTML de cualquier sitio.

Desde un punto de vista puramente técnico, es difícil describir qué es exactamente. La primera vez que te hablan del web scraping te imaginas una herramienta software perfecta que sabe lo que quieres y lo hace a la perfección, con solo darle a un botón consigue hacer tus deseos realidad. Y realmente esto no es mentira, existen herramientas poderosas que permiten hasta hacer bots que busquen por sí solos, pero eso no es lo que he venido a contar aquí. Vamos a tratar el web scraping no como una herramienta concreta, sino más bien como un arte. El arte de conocer qué datos quieres extraer de una web y utilizar un lenguaje de programación para crear lo que tú quieras hacer. En definitiva, de ser capaz de crearte algo a medida, y tratándolo siempre desde un enfoque más de autoaprendizaje e investigación que desde un enfoque de producto.

No obstante, el hecho de tratar este tema desde un punto de vista investigador no es incompatible con la posibilidad de crear, con lo aprendido, cualquier sistema de mayor o menor complejidad que sea capaz de resolver un problema en este ámbito. De hecho, como conclusión a este trabajo, la idea inicial es crear un sistema visual e intuitivo que pueda ayudar a cualquier usuario tanto a realizar extracciones básicas de datos en webs mediante las técnicas que estudiaremos como a iniciarse y dar sus primeros pasos en este mundo.

Objetivos, metodología y estructura

El objetivo único que guiará toda la intencionalidad de esta obra es la investigación. No se trata de publicar nada novedoso ni de crear un producto disponible para el mercado.

La primera parte se trata de conocer el estado actual de todo lo que gira alrededor del web scraping. Tecnologías, empresas de servicios, intencionalidad de usuarios, aplicaciones en marcos prácticos, etc. Con todo ello, elaborar cuando corresponda pruebas de concepto que puedan aportar valor a la hora de obtener conclusiones sobre este ámbito.

Con un buen estudio previo podemos conocer qué actores participan en este marco, qué tecnologías están implementadas y cuáles están más o menos maduras, así como qué buscan obtener los usuarios mediante estas técnicas y qué valor aportan al mercado tecnológico actual. Se tratará, en la medida de lo posible, de buscar qué problemas todavía están por resolver o qué debilidades existen en la actualidad. Conociendo las debilidades actuales,

podemos idear más fácilmente caminos claros de nuevos desarrollos y tecnologías, siendo primordial para la tercera parte.

La última parte consistirá en analizar lo recopilado anteriormente para tratar de avanzar hacia un punto en concreto. Si se da el caso, la intención es crear una herramienta que haga uso de algunas tecnologías previamente analizadas que cumpla dos requisitos:

- **Servir como guía de aprendizaje:** para ello es esencial escribir buen código. Que la herramienta sirva para que otros encuentren en ella una forma de apalancamiento de aprendizaje, una forma más clara de ver el concepto y poder probarla para ver lo que hace y entender el código para ver cómo lo hace.
- **Facilitar la recopilación de datos** como herramienta útil. No se trata de crear un buen producto de mercado, pero ¿por qué no aglomerar diversas tecnologías para poder acceder a toda su funcionalidad desde una única herramienta?
- **Usar dicho proyecto de desarrollo como puente** para obtener conocimiento en otras herramientas y metodologías como TDD, integración continua, sistemas de virtualización por contenedores, gestión de caché y bases de datos en memoria, métricas de código, patrones, principios SOLID, etc.
- **Crear buen código**. Por ser el cuarto objetivo no es para nada menos importante. Un buen código entendible, modularizado, mantenible, escalable, con baja deuda técnica (y aquí paro, pero la lista es muy larga) es crucial para poder cumplir otros objetivos como la utilidad del proyecto como guía de aprendizaje o como herramienta útil para otros usuarios y/o programadores futuros.
- **Crear buena documentación**. Totalmente ligado al objetivo anterior. Sin documentación adecuada que aclare la estructura del proyecto, explique su funcionamiento y clarifique sus casos de uso, los demás objetivos terminarán cojeando.

Estado del arte

A continuación, analizaremos la actualidad de este campo. Será un análisis teórico-práctico con el que se pretende abarcar todo el ancho posible de este campo, profundizando en aquello que suponga un mayor interés para el avance de la obra.

El análisis tendrá dos aspectos claramente marcados:

- **Ámbito objetivo:** recopilación de datos y fuentes, pruebas de concepto, datos comparativos entre herramientas, etc.
- **Ámbito subjetivo:** basado en el ámbito objetivo, ¿qué partes son realmente interesantes para profundizar? ¿Hacia dónde se desea guiar este trabajo? ¿Qué elementos o actores serán descartados para centrarse más en aquellos de mayor relevancia? Como en todo trabajo, siempre existe una parte subjetiva. En este caso, en base a todo lo recopilado de ámbito objetivo, el autor podrá avanzar en base a sus preferencias personales o conocimientos previos, así como las condiciones propias del entorno marcarán también metas posiblemente realizables y metas limitantes que, por condiciones externas, no sería posible o no conviene profundizar.

Atendiendo a estos dos ámbitos que van a regular todo el avance, como autor debo aclarar, previamente al análisis, que el hecho de seleccionar una tecnología entre varias o una metodología, así como cualquier otro ámbito de decisión que venga definido por mis propias preferencias o limitaciones del entorno, no significa que todo aquello desechado sea de peor calidad o menor interés, siendo yo, el autor, el responsable de evocar la obra hacia donde considere más oportuno y poder obtener así un final más satisfactorio.

Primeras búsquedas

Si buscamos *web scraping* en Google, la mayoría de los enlaces en la primera página se trata de enlaces informativos acerca de qué es o cómo usarlo.

Encontramos un artículo interesante en *papelesdeinteligencia.com* sobre 10 herramientas comunes para extraer datos automáticamente[1], y nos da una buena base para comenzar a ver cómo está un poco por encima el mundo de las herramientas para el web scraping. Está

[1] https://papelesdeinteligencia.com/herramientas-de-web-scraping/

especialmente pensado para no programadores, lo que permite acercar este mundo a gente menos técnica.

Existe un artículo similar[2], esta vez escrito en inglés de *scraperapi.com*, que menciona otras 10 herramientas. Ambos artículos coinciden solo en 3 herramientas, por lo que se obtienen 17 nombres de herramientas que pueden ser analizadas y/o probadas.

Herramientas de web scraping

Son muy numerosas. A continuación, se citan en lista las herramientas para extracción de datos encontradas con relevancia para la obra, ordenadas alfabéticamente:

Apify.com

Es altamente recomendado conocer bases de JavaScript para poder utilizar esta herramienta. Al igual que *webhose* (definida más adelante), permite conectarse a una API y obtener los datos en CSV, JSON, XML, RSS, etc. Tiene plan gratuito y ofrece también planes de pago que parten de los 19 dólares mensuales, lo que no la sitúa para nada en los puestos de las más caras.

	Developer Free	Freelancer $49/month	Startup $149/month	Business $499/month	Enterprise
Actor compute units	10	100	400	2000	∞
Data retention	7 days	14 days	21 days	30 days	∞
Shared datacenter proxies	30 (one month trial)	30	100	400	∞
Residential proxy traffic	200 MB	200 MB	Custom	Custom	∞
Google SERPs	100	100	Custom	Custom	∞
Support	Community	Email	Email	Email	Priority Private Slack channel
	Sign up	Subscribe	Subscribe	Subscribe	Contact us

Según muestran en su web, tienen dos ramas principales. Una más pensada para desarrolladores que pretenden obtener datos de webs en bruto y otra más enfocada a capas

[2] https://www.scraperapi.com/blog/the-10-best-web-scraping-tools

de negocio, donde lo importante es la capacidad de representación e interpretación de la información para poder analizarla y proceder a una toma de decisiones adecuada. Es decir, un nivel de consultoría de datos propiamente dicho.

Disponen de documentación de calidad en la web y de ejemplos de casos de estudio. Es posible, siguiendo sus tutoriales, poder analizar campañas como el Black Friday en tiendas online concretas sin tener apenas conocimientos.

No dispone de una interfaz gráfica altamente utilizable, está más pensada para programadores, ya que su servicio básico es una librería utilizable a nivel de API. Lo que sí dispone son herramientas más visuales que permiten codificar un crawler básico en pocos segundos. Incluso es posible recrear una prueba de concepto desde su página web escrapeando 5 páginas de un sitio determinado.

Beautiful Soup

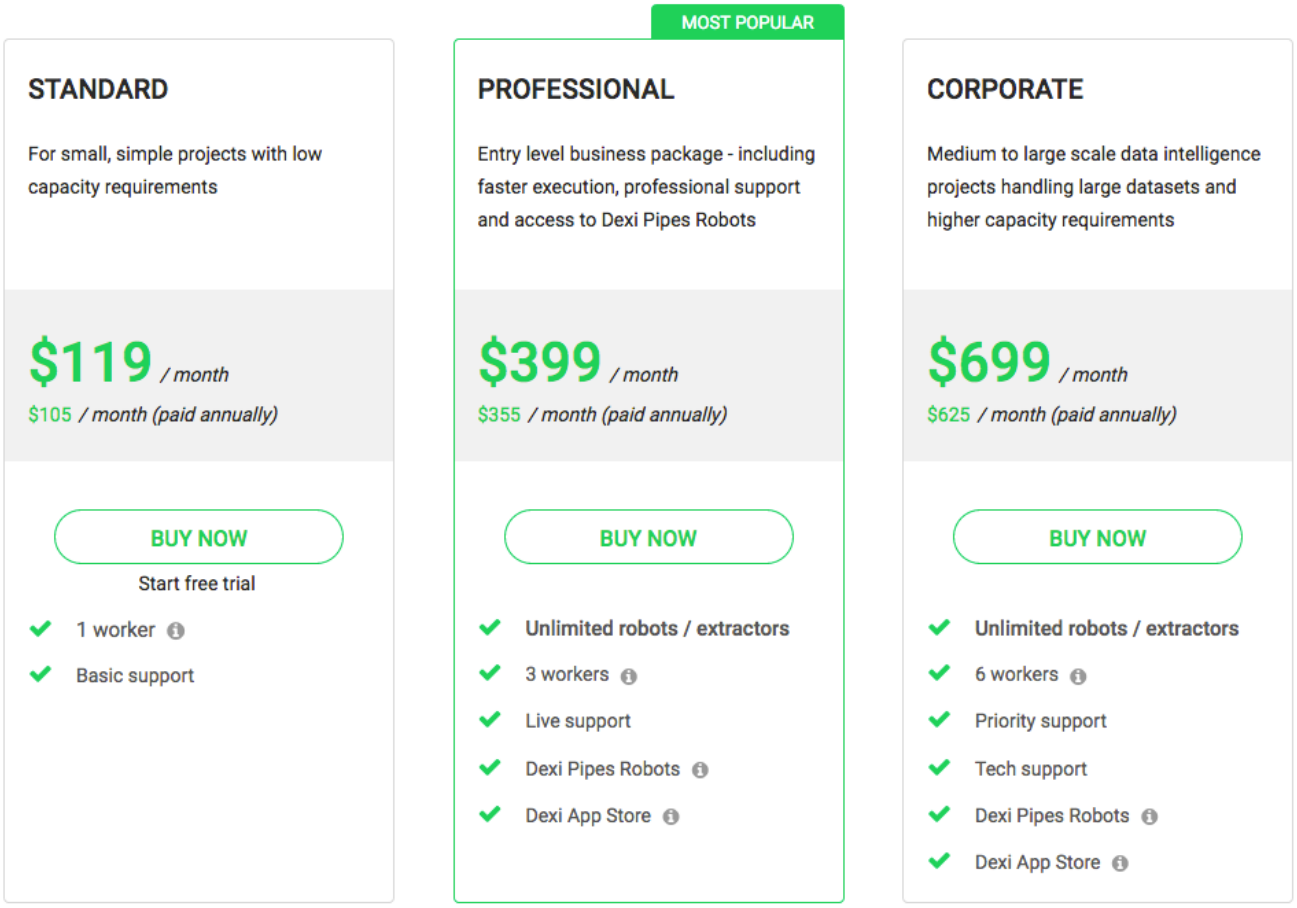
Una librería de Python con todas las ventajas que ofrece el hecho de ser una simple librería: gratis, uso ilimitado y totalmente configurable y personalizable. Ahora bien, con todas sus desventajas: como no sepas Python la curva de aprendizaje se multiplica, tú mismo tienes que hacértelo todo y estás limitado a sistemas locales ya que, si lo instalas en arquitecturas cloud, pierdes la ventaja de la gratuidad.

Ahora bien, si la comparamos con Scrapy (la librería por excelencia para este contexto), sí tiene más facilidad de uso. Con Beautiful Soup encontramos la potencia y flexibilidad que ofrece una librería de este tipo junto a una curva de aprendizaje menor que otras como Scrapy, renunciando también a parte de su potencial y centrándose en usuarios que buscan ciertas cosas en concreto, facilitando así el acceso a ella.

Dexi.io

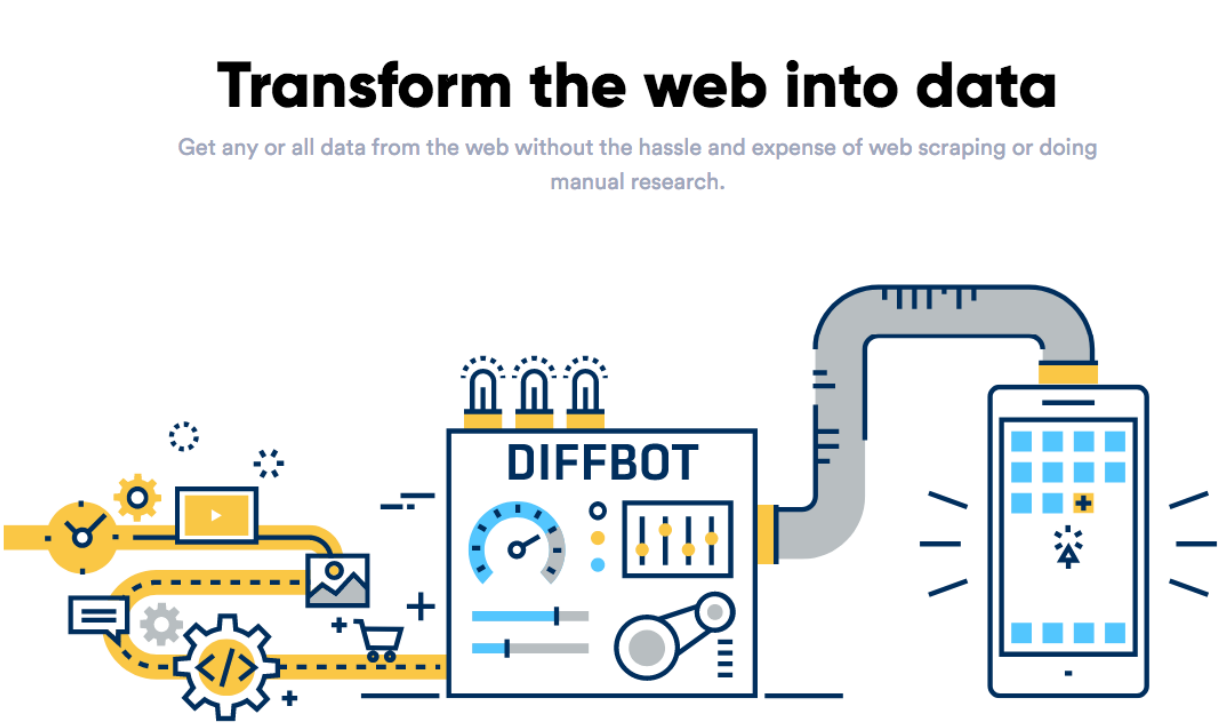
No dispone de plan gratuito de prueba. Permite utilizar Arañas y bots, entre otros sistemas. No obstante, su curva de aprendizaje es algo más elevada que otras herramientas. Es por eso que en algunos sitios, como en el artículo mencionado de *papelesdeinteligencia*, la definen como *"la herramienta de web scraping para usuarios avanzados".*

Aun así, a pesar de su alta complejidad, sí tiene ciertos apoyos basados en herramientas visuales que permiten organizar la arquitectura de un crawler a nivel visual. El gran potencial de esta herramienta es que sus bots no se dedican solo a la extracción recursiva y pura de datos, sino que pueden ser automatizados para realizar tareas sencillas y profundizar mejor en los datos deseados.



Diffbot.com

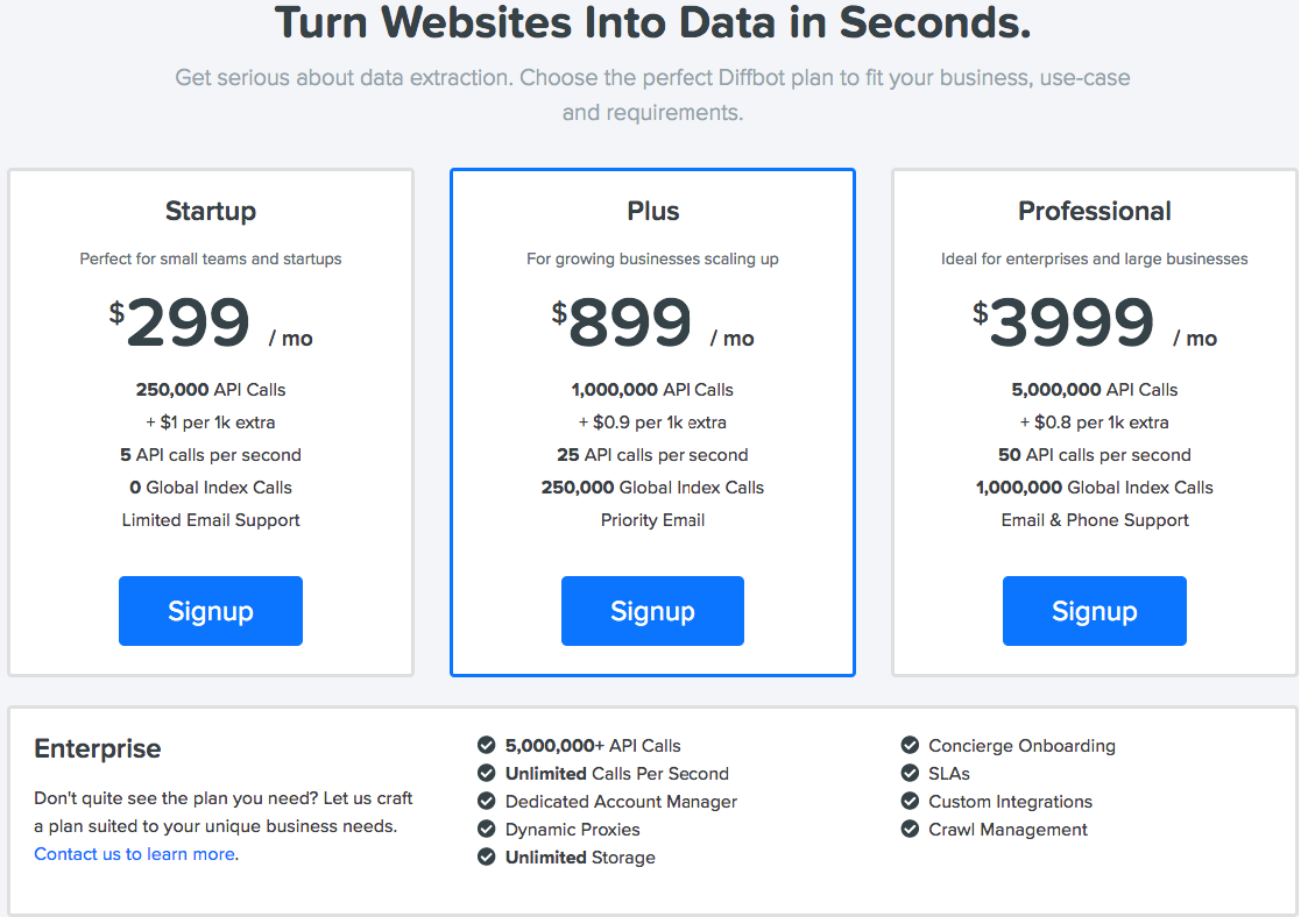
Como ellos mismos expresan en el titular de su web, su mayor aporte de valor consiste en la utilización de Inteligencia Artificial para mejorar la capacidad de filtración y extracción de datos.



Ponen a disposición del cliente diferentes APIS prediseñadas que permiten obtener diferentes datos de diferentes tipologías web: foros, artículos, etc.

DIFFBOT

Prueba de 14 días gratuita pero, después, los precios más baratos parten de los 300 dólares. Su potencial radica en que, con ayuda de la inteligencia artificial, puede mejorar la búsqueda de elementos sin estar limitados a simples búsquedas por estilos o querys de elementos.



Hunter.io

Podemos considerarlo como una herramienta de extracción de datos, pero centrada en correos electrónicos. Por tanto, sale de la temática de "extracción de datos en páginas web" hacia la que está pensada esta obra.

Se trata, básicamente, de un buscador de emails en dominios personalizados. Útil si quieres contactar con alguna compañía.

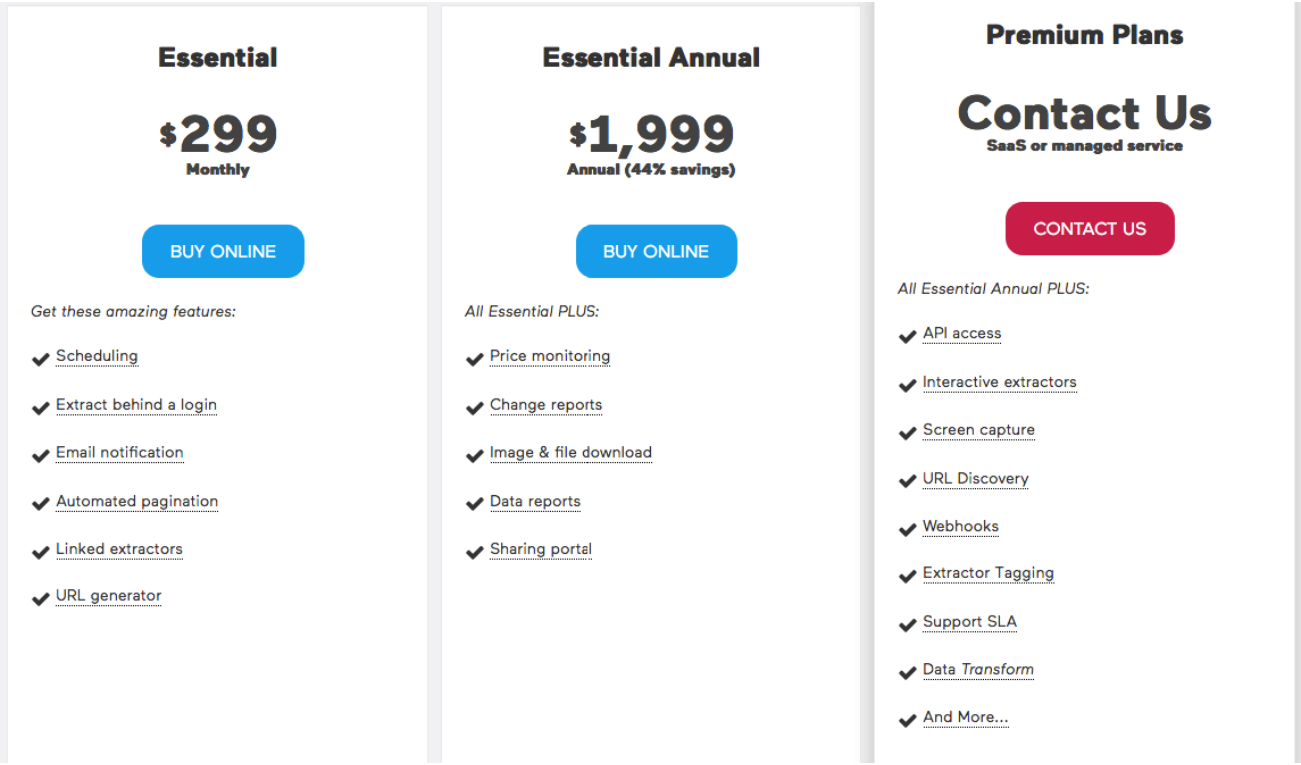
La búsqueda puede ser en ambos sentidos. Es decir: podemos buscar correos de un dominio concreto (@dominio.com) o también podemos buscar un nombre de usuario y ver en qué dominios está adscrito un correo electrónico bajo ese alias, aunque esta segunda no funciona del todo bien.

Import.io

Una de las más utilizadas para ejemplificar el concepto de web scraping. Muy recomendada desde sitios como *ecomaster*[3] para analizar a la competencia de un entorno e-commerce.

Dispone de una interfaz de usuario amigable y fácil de utilizar. Realmente es buena pero, su mayor pega y la que más limita su número de usuarios, es que es muy cara: entre 300 dólares mensuales o 2000 anuales para los planes básicos. Posee versión gratuita, pero solo dura 48 horas, lo que la convierte en mala candidata para realizar investigaciones y pruebas de concepto.

[3] https://ecommaster.es/analisis-de-competencia-import-io

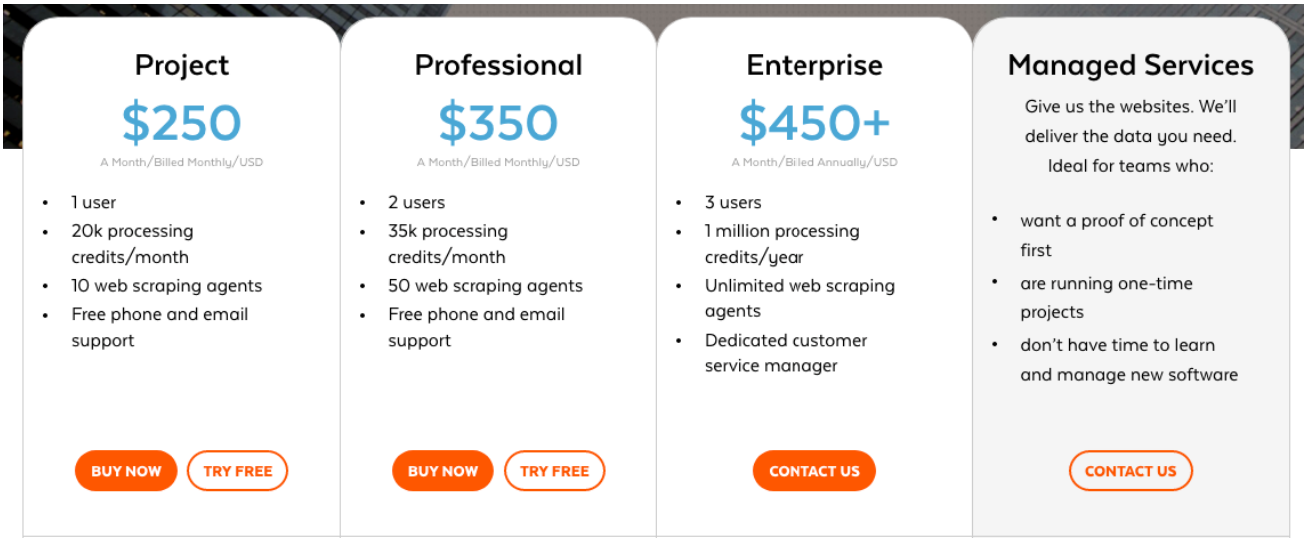


Y sí, también permite utilizar algoritmos de Machine Learning para automatizar la recolección de datos.

Mozenda.com

Definida como la mejor opción para las empresas que buscan una plataforma basada en la nube. Tienen almacenada gran cantidad de información de todas las páginas que han *escrapeado*, por lo que se permiten posicionarse, además, como una empresa DAAS (Data As A Service). Por tanto, su negocio no va enfocado exclusivamente a ofrecer una herramienta útil de extracción de datos, sino que es una de sus ramas.

Descartada para pruebas de concepto, ya que debes pagar una licencia mínima de 250 dólares.



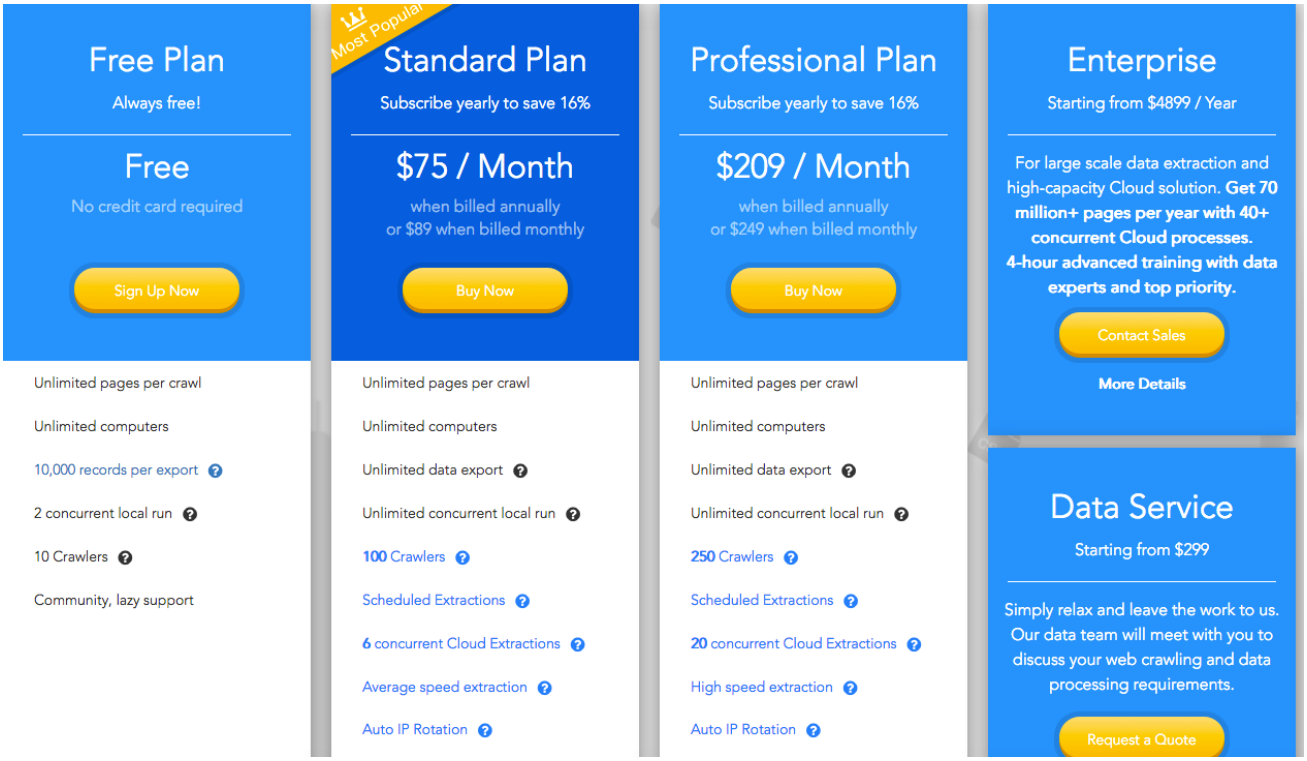
Octoparse

La gente de ScraperApi los definen como *una buena herramienta para gente que busca extraer datos de sitios web sin tener que codificar nada*.

Los planes de pago no son los más baratos, pero tampoco son abusivos. Parten desde los 75 dólares pero, ojo al dato, tienen un plan gratuito ilimitado muy generoso.

Octoparse

Lo que buscan ofrecer, ante todo, es buena experiencia de usuario. Como dicen en su propia web "*point, click and extract*".

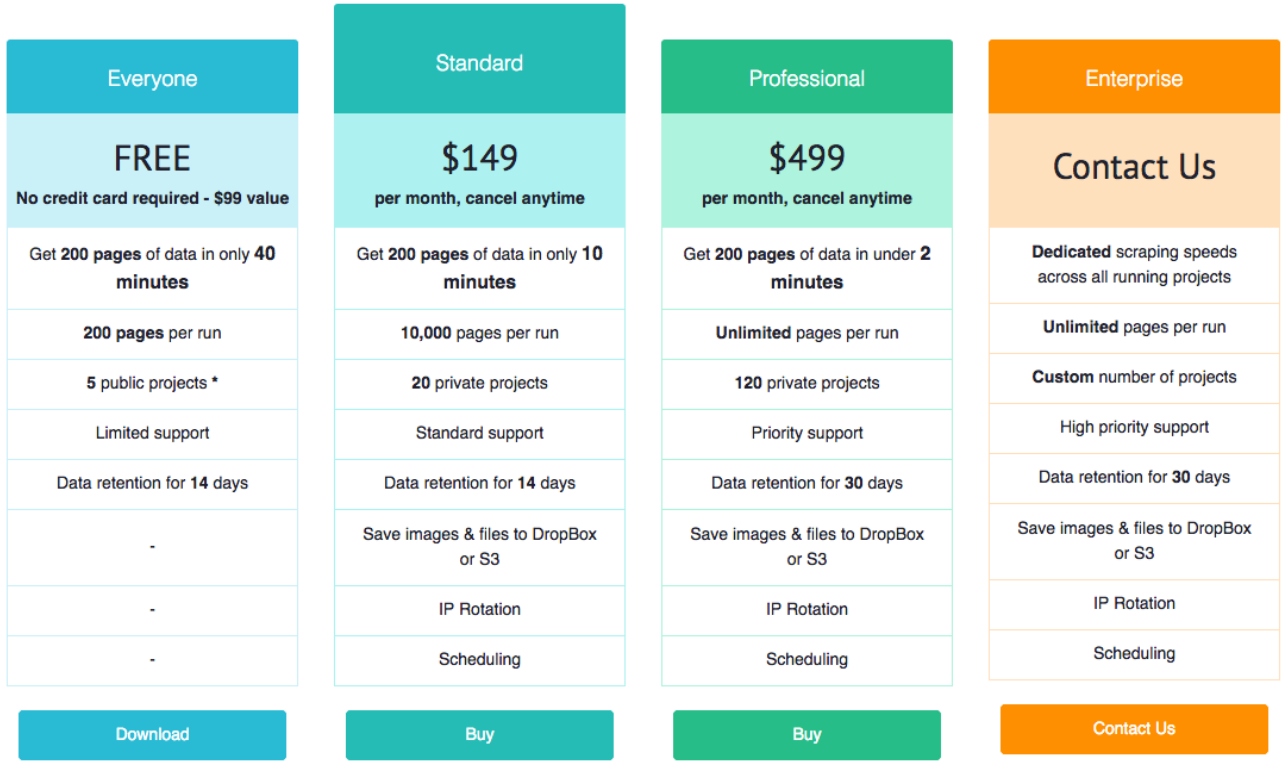


No permite configurar muy a bajo nivel el algoritmo del crawler, así que, seguramente, funcionará mejor en unos sitios que en otros. Pero, sin duda, con ese extra de usabilidad que ofrece, tienen las puertas abiertas a gran cantidad de público.

ParseHub

Buena opción para crear crawlers sin codificar en ningún lenguaje. Puede ser utilizado por alguien con altos conocimientos técnicos (como un Data Scientist) o por perfiles menos profundizados como periodistas o analistas de datos. De esta forma, permite que el usuario final se centre en lo importante: tratar los datos. Sin perder excesivos recursos en su obtención. Además, hablan bien de su plan gratuito, lo que lo convierte en muy buena opción para probar en un entorno de investigación y aprendizaje como este.

Realmente, no podemos decir que consigan un *point, click and extract* como Octoparse. Sino que, más bien, toda la personalización del crawler puede llevarse de forma visual. Dotándolo de mayor potencia y usabilidad, pero también lastrando la simpleza.



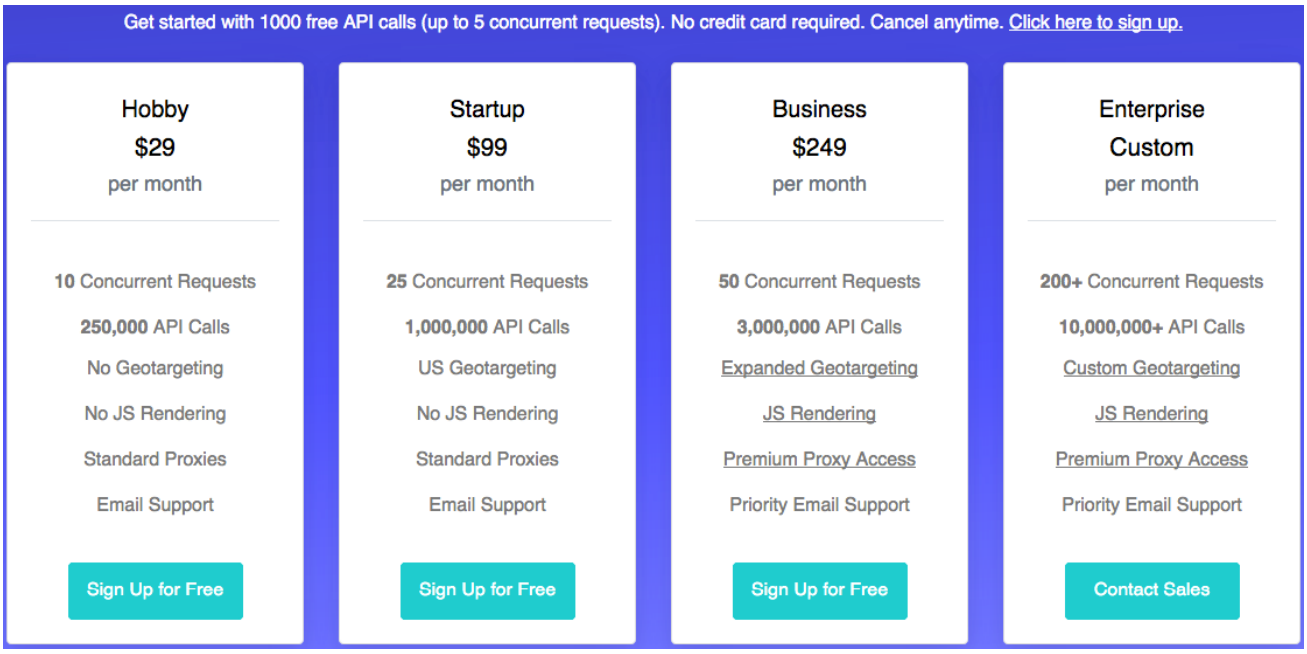
Everyone	Standard	Professional	Enterprise
FREE **No credit card required - $99 value**	$149 **per month, cancel anytime**	$499 **per month, cancel anytime**	Contact Us
Get **200 pages** of data in only **40 minutes**	Get **200 pages** of data in only **10 minutes**	Get **200 pages** of data in under **2 minutes**	**Dedicated** scraping speeds across all running projects
200 pages per run	**10,000** pages per run	**Unlimited** pages per run	**Unlimited** pages per run
5 public projects *	**20** private projects	**120** private projects	**Custom** number of projects
Limited support	Standard support	Priority support	High priority support
Data retention for **14** days	Data retention for **14** days	Data retention for **30** days	Data retention for **30** days
-	Save images & files to DropBox or S3	Save images & files to DropBox or S3	Save images & files to DropBox or S3
-	IP Rotation	IP Rotation	IP Rotation
-	Scheduling	Scheduling	Scheduling
Download	Buy	Buy	Contact Us

ScraperApi

Esta herramienta se clasifica como herramienta para desarrolladores. Permite crear *scrapers* para analizar la WEB, gestionar proxies e incluso CAPTCHA. Utiliza una API, por lo que su uso es bastante sencillo. Eso sí, no es un servicio de *datafiniti*, es decir, no ofrecen datos ya parseados de forma estructurada a través de una API, sino que utilizas las APIs para controlar tus crawlers.

Es una herramienta bastante popular y goza de buen estatus en la comunidad Open Source, ya que colabora bastante. Tiene muy buena documentación disponible y fácil de encontrar. Permite iniciar sesión automáticamente con los servicios de Google y con las credenciales de GitHub.

Tiene precios bastante competitivos, partiendo de los 29 dólares. Además, puedes hacer las primeras mil llamadas a la API de forma gratuita.



Pero no todo lo bueno se acaba aquí. Como podemos ver en la figura a continuación, si tus llamadas a la API son para fines de investigación y testing, puedes contactar con ellos para solicitar más de forma totalmente gratuita. Es por estas razones por las que goza de buena fama.

FAQ

DO YOU HAVE A FREE PLAN?

You can sign up here and get 1000 free API calls (with a maximum of 5 concurrent connections), if you need additional API calls for testing purposes, please contact support

Otra de las razones por las que goza de buena fama en la colaboración con la comunidad es por su buena política competitiva. No solo en términos de precios. Su artículo *"The best data scraping tools and web scraping tools"* [4], que ha sido una importante fuente para poder elaborar esta lista de herramientas, muestra su transparencia ya que, en su mismo sitio web, ofrecen una lista con las herramientas que ellos consideran las mejores para que el usuario pueda evaluar cuál es la mejor para desempeñar la tarea que busca. Es cierto que ScraperApi es la primera de esta lista, pero creo que podemos permitirles ese lujo, ¿no?.

Scrapy

En el mundo de los desarrolladores, podemos considerar Scrapy como la librería por excelencia. Totalmente gratuita y extremadamente potente. Eso sí, es obligatorio saber Python y tú tienes que pegarte con todo a lo bruto.

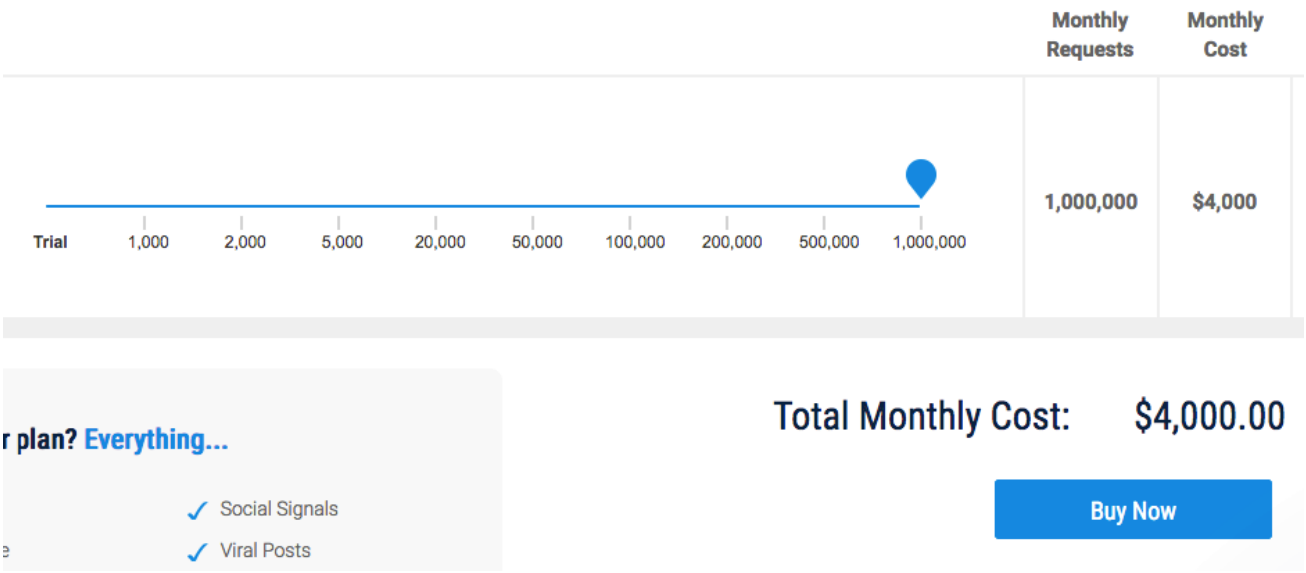
[4] https://www.scraperapi.com/blog/the-10-best-web-scraping-tools

Webhose.io

Para los perfiles más técnicos, *webhose* se convierte en una muy curiosa alternativa. Tienen sus propios e*scrapeadores* de datos y un servicio de estructuración que permite, sin tener que ser tú mismo el que escrapea diversos sitios, obtener datos estructurados obtenidos de sitios no estructurados conectándose a una API, a la que puedes hacer 1000 peticiones al mes gratuitas. Así, es fácil obtener los datos en formatos estandarizados y estructurados como XML o JSON.

Resulta muy curiosa para probar. La capacidad de poder obtener los datos deseados de manera estructurada a través de una API hace que los perfiles más técnicos nos enamoremos de ella a simple vista. Ahora bien, su punto fuerte puede ser también su tendón de Aquiles: es necesario tener conocimientos medios / altos en diversos lenguajes y tecnologías para poder consumir su API. Y, además, presenta el mismo problema que toda las APIS del universo: serán más usables en la medida que su documentación sea más o menos buena.

La tarificación que ofrecen se basa en la cantidad de peticiones que haces a su API. Te ofrecen 1000 peticiones gratuitas mensuales, llegando a los 4000$USD si haces un millón de llamadas.



80legs.com

Ellos mismos se definen como *"una herramienta de scraping fácil"* ofreciendo los dos servicios básicos que son clave en este campo:

- **Crawling customizado:** puedes configurar tus propios *escrapeadores* y extraer datos como a ti te interese.
- **Datafiniti:** *"Omite el scraping y obtén acceso instantáneo a los datos de la web"*, citan ellos mismos en su web. Como otras ya mencionadas, ofrecen capacidad de acceder a una api para tener acceso instantáneo a datos sin perder el tiempo parseando sitio, de eso ya se encargan ellos.

Poseen un gran plan gratuito y podemos ver que no son precisamente los más caros del mercado. Además, ofrecen servicios (facturados aparte) en los que son ellos los que crean

el crawler a medida si no sabes hacerlo y buscas unos datos específicos que no están todavía en su otra empresa, *Datafiniti*.

Criterios generales de comparación

Para poder extraer conclusiones del conjunto de herramientas mencionadas en la lista, se atenderán algunos criterios concretos para poder analizar más cuantitativamente el abanico de productos y servicios tanto de cada empresa a nivel individual como de lo que ofrecen todas ellas conjuntamente.

Crawling personalizado

Evaluaremos qué herramientas ofrecen productos software o servicios de scraping personalizado. Es decir, si el usuario puede, de forma más o menos sencilla, elaborar un crawler con las características específicas deseadas.

No entran aquí servicios de oferta de crawlers predefinidos, tampoco son válidos los servicios de Datafiniti o DAAS y tampoco contarán servicios que separen la ejecución del control del usuario. El crawler creado por el usuario debe permanecer bajo su control total y los datos recopilados serán devueltos al mismo, independientemente de si son tratados o no para ser añadidos a contenedores de datos para DAAS.

Criterio de clasificación binario: ¿ofrece o no ofrece el servicio?

Interfaz gráfica

Independientemente de si ofrece una mejor o peor experiencia de usuario, clasificaremos las diferentes marcas en base a su oferta de interfaces gráficas que den un servicio familiar para prácticamente cualquier usuario sin necesidad de codificar o aquellas que, al contrario, se basen exclusivamente en ofrecer productos para un público de alto nivel técnico.

En este apartado nos olvidaremos de comparar potencia, casos de uso y/o escalabilidad. Evidentemente, por norma general, cuanto más intuitiva sea una herramienta, más limitadas serán sus opciones de uso. Por otro lado, aquellas que no ofrecen ninguna clase de facilidad al usuario no técnico, suelen gozar de mayores casos de uso, mayor potencia y mayor escalabilidad.

Criterio de clasificación binario: ¿tiene o no interfaz gráfica de usuario?

Machine Learning / IA

Algunas de las empresas mencionadas incluyen la posibilidad de mejorar los algoritmos de búsqueda utilizando análisis con Machine Learning e, incluso, Inteligencia Artificial. Normalmente, como es de esperar, este servicio encarece mucho el precio final del servicio.

Criterio de clasificación binario: ¿tiene o no servicio de ML/IA aplicado al web scraping?

Planes gratuitos

Conocer cuáles tienen planes gratuitos. Esta parte de la clasificación resulta algo más complicada, ya que existen algunas herramientas que, si bien ofrecen plan gratuito, es simplemente una demo muy limitada en funciones, tiempo o cantidad de uso.

Por ello, esta parte será evaluada bajo un **criterio de clasificación ternario**: existe, no existe o existe limitado.

Consideraremos que es limitado cuando dicho plan gratuito no sea suficiente ni siquiera para pruebas de concepto o estudios para obras de investigación, como esta. Una herramienta que ofrezca todo su servicio durante solo 48 horas, por ejemplo, la consideraremos limitada. También consideraremos limitada aquella que limite su funcionalidad en un plan gratuito de forma que aquello que más interés tenga para esta obra no sea accesible.

Un servicio de plan gratuito con las funcionalidades deseadas para este estudio, sin límite de tiempo (o superior a un año) pero, por ejemplo, con un límite de URLs scrapeadas o llamadas a APIs superior a 1000 sería considerado como SÍ, TIENE PLAN GRATUITO; ya que este plan sería suficiente para poder elaborar una prueba de concepto y ser comparada con otras.

Datafiniti / DAAS

Definimos este servicio como la capacidad, por parte de las empresas, de ofrecer un tratamiento directo y estructurado de los datos; separando así el web scraping del data mining.

Normalmente, cuando una empresa cuenta con este servicio, ofrece al usuario acceso a una API que le permite solicitar la información deseada y la retorna en formatos ya estructurados, generalmente JSON.

El usuario no se preocupa de cómo se obtienen esos datos. Tampoco debe perder tiempo configurando un crawler o esperando a que se ejecute y obtenga resultados, sino que debe

mandar una instrucción predefinida al sistema y éste debe devolverle el conjunto de datos solicitados de manera directa. Por supuesto, resulta evidente que puede haber intervalos de espera y/o sincronización debido a la cantidad ingente de datos que pueden llegar a solicitarse o que deben ser explorados para extraer aquellos deseados por el usuario. Pero una cosa es buscar los datos en un set ya definido y otra cosa es recolectarlos, ordenarlos y devolverlos.

Criterio de clasificación binario: ¿ofrece o no ofrece servicio DAAS?

Documentación pública de alta calidad

Se evaluará la disponibilidad de documentación que permita reducir los tiempos de aprendizaje y puesta en marcha de sistemas para la recolección de datos usando cada herramienta.

Debido a la importancia de este apartado, no podemos simplemente observar si tiene o no (además, seguramente sí tenga) sino que es vital verificar si es una buena documentación o si es una documentación básica que no permite resolver los problemas que se encuentran durante la instalación y puesta en marcha de cada sistema.

Por tanto, se trata de un **criterio de clasificación ternario:** ¿Es la documentación de buena calidad, mala o inexistente?

Conjunto de planes de pago

Se trata de evaluar, en unidades estandarizadas (€/mes), el importe de los planes más baratos y de aquellos más caros. Olvidando el resto de planes intermedios, para poder crear una idea de hasta dónde puedes llegar en cada sistema.

Junto a la evaluación de los planes gratuitos, se convierte en una buena medida de accesibilidad del público general a cada sistema.

No se trata de un criterio binario o ternario, simplemente se trata de **comparar cantidades y tamaños de abanicos económicos**.

Comunidad y repositorios públicos gratuitos

Un aspecto actual en gran cantidad de sistemas y tecnologías es la posibilidad de compartir creaciones propias con otros: github, thingiverse, gitlab, bancos de imágenes, etc.

Algunas entidades que ofrecen los servicios estudiados incluyen también la posibilidad de compartir los crawlers creados a disposición del público y, por supuesto, poder utilizar los ya creados previamente por otros. De esta forma, se permite que los actores se apalanquen entre ellos y mejore con creces la evolución de la extracción de datos de cada sitio, pudiendo ser explorado en conjunto desde múltiples puntos de vista que no guardan para sí mismos las herramientas y las comparten.

Criterio de clasificación binario: ¿Ofrece o no servicio de repositorios públicos gratuitos?

Matriz general de clasificación

Tras este estudio, se ha deducido la siguiente tabla que muestra los elementos, herramientas, precios y servicios que ofrecen las diferentes marcas de productos y servicios.

NOMBRE / WEB	CRAWLING PERSONALIZADO	INTERFAZ GRÁFICA	ML / IA	PLAN GRATUITO	DATAFINITI / DAAS	REPOSITORIOS PÚBLICOS	DOCUMENTACIÓN DE CALIDAD	PLANES DE PAGO	
								MÁS BARATO	MÁS CARO
Apify.com	SI	NO	NO	SI	NO	SI	SI	49 $/MES	499 $/MES
Beautiful Soup	SI	NO	NO	SI	NO	NO	MALA	0	0
Dexi.io	SI	SI	NO	NO	NO	NO	SI	119 $/MES	699 $/MES
Diffbot.com	SI	NO	SI	LIMITADO	SI	NO	SI	299 $/MES	3999 $/MES
Hunter.io	NO	NO	NO	LIMITADO	NO	NO	SI	34 $/MES	399 $/MES
Import.io	SI	SI	SI	NO	SI	NO	SI	299 $/MES	1999 ++ $/AÑO
Mozenda.com	SI	SI	NO	NO	SI	NO	SI	250 $/MES	450+ $/MES
Octoparse	NO	SI	NO	SI	SI	NO	MALA	75 $/MES	4899 $/AÑO
ParseHub	SI	SI	NO	SI	NO	NO	SI	149 $/MES	499 $/MES
ScraperApi	SI	NO	NO	LIMITADO	NO	NO	SI	29 $/MES	249 $/MES
Scrapy	SI	NO	NO	SI	NO	NO	SI	0	0
Webhose	NO	NO	NO	LIMITADO	SI	NO	SI	50 $/2000 req	4000 $/1M req
80legs.com	SI	NO	NO	SI	SI	NO	SI	29 $/MES	299 $/MES
Jsoup	SI	NO	NO	SI	NO	NO	SI	0	0
CARACTERÍSTICAS SERVICIOS WEB SCRAPPING									

Finalizado el estudio de las diferentes empresas y herramientas, podemos concluir algunas afirmaciones:

- Las **más caras** tienden a ser aquellas que **incorporan ML/IA** en sus planes, junto con aquellas que venden **mayor usabilidad**.
- Existen empresas como Webhose y Octoparse que, si bien son tratadas en el mundo de la recopilación de datos, se dedican exclusivamente al DAAS.
- Las más baratas son las librerías de uso propio, sin prácticamente límites en cuanto a casos de uso, pero demasiado complejas de utilizar. Por tanto, **no se paga un producto, se paga un servicio y por mayor facilidad de uso**.
- Los creadores de las herramientas tienen conciencia de su uso y complementan, en general, cualquiera de ellas con **buena documentación**.

- Algunas, como 80legs, permiten descargar su software y ejecutarlo en local de forma gratuita. Una vez más, **no se vende un producto** o código de extracción de datos, **sino el servicio** de ejecución **en la nube** y de **consultoría de datos**.

Jsoup

¿por qué?

El criterio de escoger Java como lenguaje y Jsoup como herramienta concreta es totalmente subjetivo.

Normalmente, las herramientas más conocidas son en Python y dicho lenguaje es, generalmente, más apreciado para este propósito. Por tanto, para intentar dotar de mayor originalidad a esta obra, se ha elegido Java (menos popular) y Jsoup como librería para tratar de investigar en campos menos explorados y poder proporcionar algún conocimiento novedoso.

Cómo usarlo (importar Maven)

Se va a utilizar Maven para gestionar y construir el proyecto.

Primero, creamos un proyecto Maven. Puede utilizarse el arquetipo que mejor convenga para compatibilizar el proyecto con otros frameworks como Spring o similares.

Para poder usar Jsoup debemos importarlo en el POM, como cualquier proyecto Maven:

```
<!-- https://mvnrepository.com/artifact/org.jsoup/jsoup -->
<dependency>
	<groupId>org.jsoup</groupId>
	<artifactId>jsoup</artifactId>
	<version>1.11.3</version>
</dependency>
```

Una vez importado por Maven, ya podemos utilizar todo lo que queramos de esta librería.

Pruebas de concepto

En este apartado se desarrollarán ejemplos de algoritmos con Jsoup, la librería elegida para ejecutar el sistema de webscraping, basados, entre otros, en la documentación oficial de Jsoup. Con esto se busca adquirir los conocimientos básicos de su funcionamiento para comprender posteriormente cómo poder integrarlo en otros sistemas.

De cadena de texto a HTML

Cargando documento completo

Dado un contenido HTML en una cadena de texto (String), es posible que necesitemos analizarlo para varios objetivos:

- Analizar y obtener su contenido de forma estructurada
- Verificar si está bien formado
- Modificarlo

Para esta labor, Jsoup incorpora un método estático: *parse()*. Este método permite instanciar un nuevo objeto de tipo *Document* con el contenido de la cadena dada, e incluso permite definir una URL base para tratar los directorios relativos y absolutos.

Para comprender este ejemplo creamos una clase sencilla:

```
package com.urbanojvr.jsoupproofs.htmlparser;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;

public class HtmlParser {

   private Document doc;

   public  HtmlParser(String strToParse){
       this.doc = Jsoup.parse(strToParse);
   }

   public  HtmlParser(String strToParse, String baseURL){
       this.doc = Jsoup.parse(strToParse, baseURL);
```

```
    }

    public Document getDoc() {
        return doc;
    }

    public void setDoc(Document doc) {
        this.doc = doc;
    }
}
```

Podemos utilizar esta clase desde el *Main* para probar el ejemplo dado:

```
public static void main(String[] args){
    String text = "<html><head><title>Proof of concept</title></head>"
            + "<body><p>Parsing String into a HTML document</p>" +
            "<p>JSOUP PROOF OFO CONCEPT</p></body></html>";

    HtmlParser parser = new HtmlParser(text);
    Document doc = parser.getDoc();
    System.out.println(doc.body());
}
```

Para verificar el correcto funcionamiento de cada ejemplo, los tests son de gran ayuda:

```
package com.urbanojvr.jsoupproofs.htmlparser;

import org.jsoup.nodes.Document;
import org.junit.Test;

import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.equalTo;

public class HtmlParserTest {

    private HtmlParser sut;
    private String text = "<html><head><title>Parsing string to
HTML</title></head>"
            + "<body><p>Parsing String into a HTML document</p>" +
            "<p>JSOUP PROOF OF CONCEPT</p></body></html>";
    private String title = "Parsing string to HTML";
```

```
    @Test
    public void should_parse_text_string_correctly() {
        String body = "Parsing String into a HTML document JSOUP PROOF OF
CONCEPT";

        sut = new HtmlParser(text);
        Document doc = sut.getDoc();

        assertThat(doc.title(), equalTo(title));
        assertThat(doc.body().text(), equalTo(body));
        System.out.println(doc.body().text());
    }

    @Test
    public void should_save_correctly_the_base_URI() {
        String baseUrl = "http://myweb.com/category/article/subarticle-
1";

        sut = new HtmlParser(text, baseUrl);
        Document doc = sut.getDoc();

        assertThat(doc.baseUri(), equalTo(baseUrl));
        assertThat(doc.title(), equalTo(title));
    }
}
```

Gracias a esta funcionalidad, podemos convertir una cadena de texto a un *Document* propio de Jsoup y, así, utilizar el poder de la librería para analizar y extraer los datos relevantes.

Analizando un fragmento de cuerpo

En el ejemplo anterior vimos cómo tratar un documento entero de HTML. Es decir, un documento con esta estructura:

```
<html>
  <head>
     <title>Titulo del sitio</title>
  </head>
  <body>
     <h1>
     H1
```

```
      </h1>
      <p>
      Párrafo
      </p>
  </body>
  <footer>
  </footer>
</html>
```

Es posible, en un ejemplo más realista, que necesitemos analizar no un documento completo, sino que tengamos una porción codificada en HTML sin estructura previa. Para esto podemos editar el *body* de un *Document* y podremos tratarlo como tal, aprovechando la potencia de esta librería sin la necesidad de un documento HTML común completo.

Nuestra clase *HtmlParser* la modificamos añadiendo un constructor vacío para evitar dependencias con el ejemplo anterior:

```
public HtmlParser(){
   this.doc = null;
}
```

Y creamos unos métodos que permitan probar esta funcionalidad:

```
public void parseBody(String html){
   this.doc = Jsoup.parseBodyFragment(html);
}

public Element getBody(){
   return doc.body();
}
```

Ahora podemos, desde los tests, probar que editamos el cuerpo. Destacar que, al editar únicamente el cuerpo del documento HTML, el encabezado debe estar vacío:

```
@Test
public void should_parse_html_body_and_head_should_be_empty(){
   String body = "<p>Paragraph 1</p>" +
           "<p>Paragraph 2</p>";
   String expectedBody = "<body>\n" +
           " <p>Paragraph 1</p>\n" +
           " <p>Paragraph 2</p>\n" +
           "</body>";
```

```
    sut = new HtmlParser();
    sut.parseBody(body);
    Document doc = sut.getDoc();

    assertThat(doc.body().toString(), equalTo(expectedBody));
    assertTrue(doc.head().text().isEmpty());
}
```

Carga de documentos

Desde URL

Aquí es cuando empieza (un poco) la magia de un Crawler. Con este ejemplo podremos cargar todo el HTML desde una URL conectándonos a ella y descargando su contenido.

Para poder proceder, creamos una clase de ejemplo con los métodos que vamos a utilizar para descargar el contenido de una URL y comprobar sus resultados:

```
package com.urbanojvr.jsoupproofs.docloader;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;

import java.io.IOException;

public class DocumentLoader {

    private Document doc;

    public DocumentLoader(){
        this.doc = null;
    }

    public DocumentLoader(String url) throws IOException {
        this.doc = Jsoup.connect(url).get();
    }
```

```
    public Element getBody(){
        return doc.body();
    }

    public Element getHead(){
        return doc.head();
    }

    public Document getDoc(){
        return doc;
    }

    public String getTitle(){
        return doc.title();
    }
}
```

Debemos comprobar que carga el documento correctamente desde una web de prueba. La web de prueba que se va a utilizar es *http://toscrape.com/*, ya que se trata de una web pensada como pruebas para este propósito de Web Scrapping.

La primera parte que se está tratando, que es cargar desde la URL. En los tests se verificarán algunos objetivos clave que demuestran que la conexión se realiza correctamente:

- El título del sitio debe ser el adecuado
- El cuerpo no puede estar vacío
- La cabecera no puede estar vacía y solo debe contener el título del sitio
- El primer elemento <h2> del documento debe contener el valor esperado

Cabe destacar que estos tests funcionan, como puede verse en el código a continuación, conectándose al sitio apropiado. Si cambia el sitio al que se conectan los tests, deberán cambiar ciertos parámetros de los tests:

```
public class DocumentLoaderTest {

    private static final String URL = "http://toscrape.com/";
    private static final String TITLE = "Scraping Sandbox";
    private DocumentLoader sutFromURL;

    @Before
    public void before() throws IOException {
```

```
        sutFromURL = new DocumentLoader(URL);
    }

    @Test
    public void should_return_correct_title() {
        assertThat(TITLE, equalTo(sutFromURL.getTitle()));
    }

    @Test
    public void should_return_not_empty_body() {
        assertFalse(sutFromURL.getBody().text().isEmpty());
    }

    @Test
    public void head_should_not_be_empty_and_only_contains_the_title() {
        assertFalse(sutFromURL.getHead().text().isEmpty());
        assertThat(TITLE, equalTo(sutFromURL.getHead().text()));
    }

    @Test
    public void first_h2_element_on_body_should_be_expected_word() {
        String expectedWord = "Books";

        assertThat(expectedWord,
equalTo(sutFromURL.getBody().select("h2").get(0).text()));
    }
}
```

Desde un archivo local

El mismo método de carga de documentos puede ser utilizado también para cargar HTML desde un archivo de un directorio local del sistema en lugar de conectarse a un sitio web.

Para poder realizar este ejemplo, añadimos un constructor como este a nuestro *DocumentLoader*.

```
public DocumentLoader(File fileToLoad, String charset) throws IOException {
  this.doc = Jsoup.parse(fileToLoad, charset);
}
```

Con este nuevo método es posible cargar el contenido html de un archivo local.

Para poder aprovechar los mismos tests, se ha descargado el contenido HTML de la web usada en el ejemplo anterior para poder localizar el mismo documento en un archivo local y, así, lograr que los mismos tests funcionen.

Ahora, antes de cada test, deben inicializarse dos tipos de SUT (*subject under test)*:

- Sujeto de carga desde una URL
- Sujeto de carga desde un archivo local

```
@Before
public void before() throws IOException {
   sutFromURL = new DocumentLoader(URL);
   sutFromLocalFile = new DocumentLoader(fileToLoad, "UTF-8");
}
```

De esta forma, ahora duplicamos los pasos de cada test verificando ambos sujetos. Es decir, es lo mismo de antes, pero multiplicado por dos:

```
public class DocumentLoaderTest {

   private static final String URL = "http://toscrape.com/";
   private static final String TITLE = "Scraping Sandbox";
   private ClassLoader = getClass().getClassLoader();
   private File fileToLoad = new
File(classLoader.getResource("toscrape.html").getFile());
   private DocumentLoader sutFromURL;
   private DocumentLoader sutFromLocalFile;

   @Before
   public void before() throws IOException {
       sutFromURL = new DocumentLoader(URL);
       sutFromLocalFile = new DocumentLoader(fileToLoad, "UTF-8");
   }

   @Test
   public void should_return_correct_title() {
       assertThat(TITLE, equalTo(sutFromURL.getTitle()));
       assertThat(TITLE, equalTo(sutFromLocalFile.getTitle()));
   }

   @Test
```

```
    public void should_return_not_empty_body() {
        assertFalse(sutFromURL.getBody().text().isEmpty());
        assertFalse(sutFromLocalFile.getBody().text().isEmpty());
    }

    @Test
    public void head_should_not_be_empty_and_only_contains_the_title() {
        assertFalse(sutFromURL.getHead().text().isEmpty());
        assertThat(TITLE, equalTo(sutFromURL.getHead().text()));
        assertFalse(sutFromLocalFile.getHead().text().isEmpty());
        assertThat(TITLE, equalTo(sutFromLocalFile.getHead().text()));
    }

    @Test
    public void first_h2_element_on_body_should_be_expected_word() {
        String expectedWord = "Books";

        assertThat(expectedWord,
equalTo(sutFromURL.getBody().select("h2").get(0).text()));
        assertThat(expectedWord,
equalTo(sutFromLocalFile.getBody().select("h2").get(0).text()));
    }
}
```

Con estos ejemplos se ha comprobado que se comporta exactamente igual ya sea cargando un archivo de una URL externa como de un documento local guardado en el sistema de archivos.

Búsqueda de datos en documentos

En el apartado anterior ya se ha visto una breve introducción a cómo se pueden buscar ciertos elementos en un documento HTML. En los últimos tests, se buscan los elementos <h2> y se obtiene el primero para ver si cuadra con el resultado esperado. En este apartado se va a profundizar un poco más en el aspecto de extracción de los datos deseados dado un documento concreto.

Para estos ejemplos, se seguirá utilizando la web anterior, pero se usará la lista de libros que proponen en lugar de la página de inicio: http://books.toscrape.com/

Métodos DOM

Los métodos DOM (Document Object Model) son muy famosos en lenguajes como Javascript. Estos métodos permiten, de una forma programática, navegar por el contenido de un documento reduciendo el uso complejo de expresiones regulares.

Jsoup nos ofrece ciertos métodos que podemos combinar para fortalecer nuestras búsquedas:

- **getElementById(String id):** busca en el documento el elemento cuyo ID coincide con el solicitado. En este caso solo devuelve un elemento, ya que con el ID sólo puede existir un elemento que coincida.
- **getElementsByTag(String tag):** busca según el tag. Es decir: tipos de parámetros HTML como <h2>, , etc. Devuelve una lista de elementos (*Elements)* que puede ser recorrida con un bucle for (extiende de *ArrayList)* y que contiene todos los elementos, en código HTML, cuyo tag coincide con el deseado.
- **getElementsByClass(String className):** para buscar elementos de la clase dada. También devuelve una lista iterable de elementos.

Nos proporciona algunos métodos más, pero estos son los más interesantes y que usaremos para ejemplificar su funcionamiento en esta prueba de concepto.

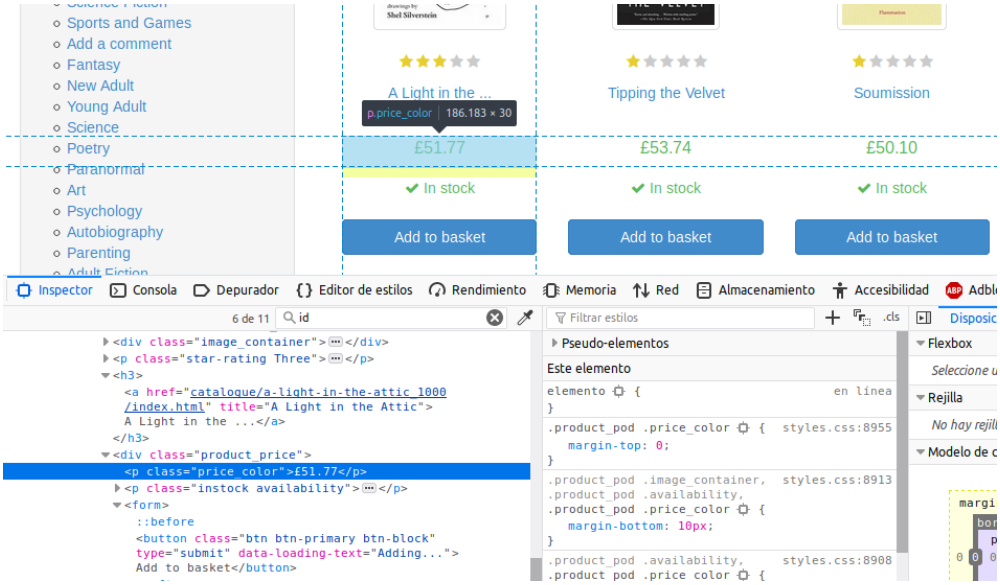
El primer ejemplo es muy sencillo. Recordemos que estamos trabajando con http://books.toscrape.com/. Partimos de una clase llamada *DomEngine* para usarla como sujeto de ejemplos:

```
public class DomEngine {

   private Document doc;
   private String charset;

   public DomEngine(String url) throws IOException {
       this.doc = new DocumentLoader(url).getDoc();
   }

   public DomEngine(File file) throws IOException {
       charset = "UTF-8";
       this.doc = new DocumentLoader(file,charset).getDoc();
   }
```

```
    public ArrayList<String> getElementTextByClass(String className){
        Elements elements = doc.getElementsByClass(className);
        ArrayList<String> pricesList = new ArrayList<String>();
        for (Element element : elements) {
            pricesList.add(element.text());
        }
        return pricesList;
    }

    public ArrayList<Element> getElementsByClass(String className){
        Elements elements = doc.getElementsByClass(className);
        ArrayList<Element> list = new ArrayList<Element>();
        for(Element element : elements){
            list.add(element);
        }
        return list;
    }

    public ArrayList<Element> getLiFromListElement(Element listElement){
        Elements elements = listElement.getElementsByTag("li");
        ArrayList<Element> list = new ArrayList<Element>();
        for(Element el : elements){
            list.add(el);
        }
        return list;
    }

    public Document getDoc(){
        return doc;
    }

    public void setDoc(Document doc){
        this.doc = doc;
    }
}
```

El primer método que vamos a probar es *getElementTextByClass(String classsName)*. Vamos a tratar de obtener la lista de los precios que aparecen en la primera página indicada.

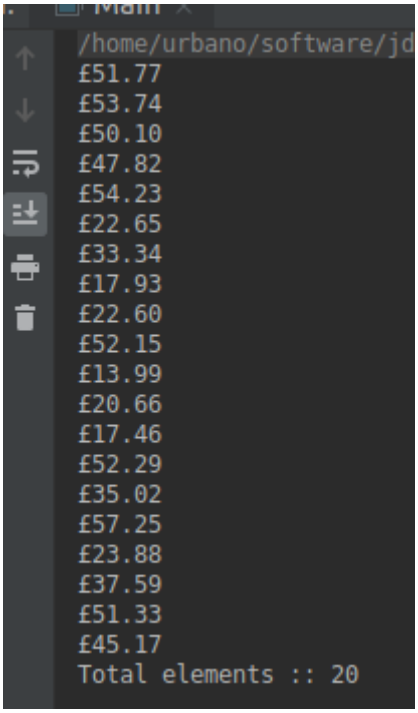
Si inspeccionamos la web, observamos que los precios están embebidos bajo esta estructura:



El elemento *<p class="price_color">* se repite en cada elemento. Por tanto, podemos usar esta información para obtener una lista de todos los precios:

```
public static void main(String[] args) throws IOException {
   String url = "http://books.toscrape.com/";
   DomEngine domEngine = new DomEngine(url);

   ArrayList<String> prices =
domEngine.getElementTextByClass("price_color");
   for(String price : prices){
       System.out.println(price);
   }
   System.out.println("Total elements :: " + prices.size());
}
```

El resultado obtenido se muestra en la siguiente imagen:



Se observa la lista de todos los precios de productos y el total de artículos en la página, 20.

Y un test de ejemplo que verifica esto puede ser:

```
public class DomEngineTest {

   private ClassLoader classLoader = getClass().getClassLoader();
   private File fileToLoad = new File(classLoader.getResource("books-
toscrape.html").getFile());
   private DomEngine sut;

   @Before
   public void init() throws IOException {
       sut = new DomEngine(fileToLoad);
   }

   @Test
   public void when_get_prices_size_should_be_20() {
       ArrayList<String> result =
sut.getElementTextByClass("price_color");
       assertThat(result.size(), equalTo(20));
   }
}
```

El test carga el archivo local, que es una copia del mismo archivo obtenido de la URL, y verifica que se ha obtenido una lista de precios cuyo tamaño es 20.

El siguiente ejemplo es un poco más complejo. Volviendo a la clase *DomEngine*, el método *getElementsByClass* llama al mismo método de Jsoup y transforma el resultado en una lista conocida. Podemos utilizar este mismo método para buscar un elemento cuya clase aparece una única vez en todo el documento. Es lo mismo, solo que nos devolverá una lista de tamaño 1.

Una vez obtenido este elemento, podemos analizarlo con el otro método que obtiene los elementos <li> de una lista, dado un elemento padre.

```
public static void main(String[] args) throws IOException {
   String url = "http://books.toscrape.com/";
   DomEngine domEngine = new DomEngine(url);

   ArrayList<Element> elements =
domEngine.getElementsByClass("side_categories");

   System.out.println("Total elements :: " + elements.size());

   elements = domEngine.getLiFromListElement(elements.get(0));
   for(Element element : elements){
       System.out.println(element.text());
   }

   System.out.println("Total elements ::" + elements.size());
}
```

Ejecutando este código en el Main, se observa cómo extraer, en este caso, el conjunto de elementos de una lista, que se trata de las categorías de libros que contiene la página en cuestión.

Finalmente, los tests de la clase *DomEngine* quedan completos de la siguiente forma:

```
public class DomEngineTest {

   private ClassLoader classLoader = getClass().getClassLoader();
   private File fileToLoad = new File(classLoader.getResource("books-
toscrape.html").getFile());
   private String className = "side_categories";
```

```
    private DomEngine sut;

    @Before
    public void init() throws IOException {
        sut = new DomEngine(fileToLoad);
    }

    @Test
    public void when_get_prices_size_should_be_20() {
        ArrayList<String> result =
sut.getElementTextByClass("price_color");
        assertThat(result.size(), equalTo(20));
    }

    @Test
    public void
when_get_class_side_categories_should_be_return_only_one(){
        int expectedSize = 1;

        ArrayList<Element> elemtns = sut.getElementsByClass(className);

        assertThat(expectedSize, equalTo(elemtns.size()));
    }

    @Test
    public void when_get_categories_list_should_be_51(){
        int expectedSize = 51;

        Element categoriesListComponent =
sut.getElementsByClass(className).get(0);
        ArrayList<Element> elementsInList =
sut.getLiFromListElement(categoriesListComponent);

        assertThat(expectedSize, equalTo(elementsInList.size()));
    }
}
```

Con esta clase se han visto algunos ejemplos prácticos de la utilidad de estos métodos.

Selección por sintaxis

Esta forma es, seguramente, más compleja de aprender. Pero, sin duda, ofrece una flexibilidad mucho mayor que utilizar métodos DOM.

Para llevar a cabo este tipo de análisis de documentos HTML, la librería proporciona el método *select()*.

Podemos utilizar este método con expresiones regulares que permiten obtener prácticamente cualquier tipo de información sin utilizar más que el método mencionado. Por ejemplo:

- Obtener elementos "<a>" con atributo "href":

```
public Elements returnAWithRef(){
   return doc.select("a[href]");
}
```

- Obtener imágenes en formato jpg:

```
public Elements getImages(){
   return doc.select("img[src$=.jpg]");
}
```

- O de cualquier formato elegido por el usuario:

```
public Elements getImages(String format){
   return doc.select("img[src$=." + format + "]");
}
```

- Buscar el de clase *container*:

```
public Element getDivContainer(){
   return doc.select("div.container").first();
}
```

- O cualquier elemento de cualquier clase:

```
public Elements searchElementAndClass(String elementTag, String
elementClass){
   String query = elementTag + "." + elementClass;
   return doc.select(query);
}
```

Y muchas cosas más. Como se ha mostrado, es posible incluso combinar diferentes elementos (como tag o clase) para hacer búsquedas muy completas.

Este método de búsqueda es, sin duda, el más potente, flexible y usable (comparándolo con los métodos DOM). Pero es cierto también que conlleva una curva de aprendizaje mayor, sin duda.

Los métodos mostrados hasta ahora como ejemplo están empaquetados en la clase *SintaxEngine*, cuyos tests se muestran a continuación:

```
public class SintaxEngineTest {

   private ClassLoader classLoader = getClass().getClassLoader();
   private File fileToLoad = new File(classLoader.getResource("books-
toscrape.html").getFile());
   private SintaxEngine sut;

   @Before
   public void init() throws IOException {
       sut = new SintaxEngine(fileToLoad);
   }

   @Test
   public void
when_get_ahref_should_return_not_empty_list_with_size_94() {
       Elements elements = sut.getAWithRef();

       assertEquals(94, elements.size());
   }

   @Test
   public void when_get_jpg_images_should_return_list_with_20_elemts() {
       Elements elements = sut.getImages("jpg");
```

```
        assertEquals(20, elements.size());
    }

    @Test
    public void when_get_png_images_should_returl_empty_list(){
        Elements elements = sut.getImages("png");

        assertEquals(0, elements.size());
    }

    @Test
    public void when_get_page_inner_div_should_be_two() {
        Elements elements = sut.searchElementAndClass("div",
"page_inner");

        assertEquals(2, elements.size());
    }
}
```

Extracción de datos

Tras extraer los elementos deseados de un documento HTML, es crucial poder extraer de ellos la información deseada.

Es posible, entre otras operaciones:

- Obtener todo el contenido del elemento en forma de texto:
    - *element.text()*
- Devolver un atributo dado:
    - *element.attr()*
- Obtener el html del elemento:
    - *element.Html();*
- Obtener el html del elemento, incluyendo sus descendientes:
    - *element.outerHtml();*

Manipulando URLs

Para completar un buen *crawler* es indispensable gestionar adecuadamente las URLs del sitio que se está analizando. La problemática no es muy amplia, pero sí conviene conocer cómo proceder en algunos casos. Cuando se trata de URLs absolutas no existe mayor problema que analizar adecuadamente el documento para extraer aquellas de verdadera relevancia, pero los problemas llegan cuando se trata de URLs relativas.

Para facilitar la labor del desarrollador en este aspecto, Jsoup incluye algunas posibilidades de regex en el método *attr()* de un atributo. Si usamos convenientemente estas reglas en los atributos <a> que contengan elementos *href* podemos crear directamente un enlace absoluto partiendo de un enlace relativo. Aunque, eso sí, la propiedad *baseURI* del documento debe estar correctamente definida. Se define automáticamente al instanciar la clase con el constructor, pero es posible editarlo si estamos analizando un análisis no dependiente de la página de inicio del sitio en cuestión.

El "crawler"

Hasta ahora se ha mencionado numerosas veces la palabra *crawler*. Literalmente, el verbo crawl significa gatear en inglés. Su sustantivación, crawler, es sinónimo de tractor.

Si nos vamos fuera del mundo del desarrollo, un crawler hace referencia, generalmente, a un vehículo todoterreno extremadamente preparado para avanzar por terrenos extremadamente escarpados campo a través. Un crawler, en el mundo de los vehículos, es lo más parecido a una persona escalando en cuanto a su capacidad de verticalidad. Esta es, podríamos decir, la definición más apropiada de crawler en español: escalador.

Un software que, dada una URL base, *escala* poco a poco todo ese sitio guardando todas las URLs obtenidas para poder analizarlas posteriormente.

Este tipo de software son la clave de los mayores motores de búsqueda del mercado, que no solo analizan todas las URLs de un sitio web, sino que analizan, literalmente, TODA la red mundial. Cuando encuentran un enlace nuevo, lo añaden a esa gran lista de enlaces. Posteriormente se procede a un análisis automatizado y a una etiquetación que permite obtener búsquedas adecuadas según lo introducido por el usuario en base a popularidad, porcentaje de coincidencia y todo lo que el SEO respecta.

Escalar un sitio web completo

Gracias a una correcta manipulación de las URLs de un sitio web, es posible no solo obtener información de una página o documento, sino escalar todo el sitio web en busca de todas sus URLs y, así, tener acceso a toda su información.

Para hacer esto es necesario poder obtener las URLs absolutas de cada enlace y, por supuesto, debe controlarse hacia dónde se está navegando, para evitar salir fuera del dominio deseado.

En la documentación de Jsoup existe un ejemplo muy claro y sencillo de cómo mostrar en una lista todos los enlaces de un documento. Además, los separa en 3 tipos:

- Links a otras páginas (URLs como tal)
- Enlaces de media (imágenes y otras fuentes)
- Importaciones (hojas de estilo y similares)

Evolucionando el algoritmo del ejemplo puede llegarse a una sencilla explicación base de cómo conseguir escalar, de forma recursiva, un sitio web concreto y obtener así todos los enlaces a su contenido. Gracias a esto, ya estaría toda esa información indexada a través de las URLs, de forma que podría ser analizada automáticamente.

Nota informativa:

- El algoritmo base está explicado en un artículo en *adictosaltrabajo.com* escrito por el mismo autor de esta obra durante el transcurso de la misma. Por tanto, todos los parecidos encontrados en el código y en la explicación del mismo son totalmente justificables, ya que ambos (esta obra y el artículo) han sido creados por el mismo autor compartiendo espacio de tiempo.

El algoritmo consiste en, primero, obtener el documento completo y analizarlo para buscar todas las URLs disponibles. Para guardar cada URL nueva en la lista, debe cumplir ciertas condiciones, como pertenecer al dominio en cuestión (contiene el *baseURI)*, ser de tipo link (los enlaces a otro tipo de contenido como imágenes o estilos no tienen relevancia) o devolver una respuesta correcta.

El algoritmo principal pintaría algo así:

```
public void crawl(String url) throws IOException {
   SintaxEngine sintaxEngine = new SintaxEngine(url);
   Elements links = sintaxEngine.getAWithRef();

   for(Element link : links){
       String actualUrl = link.attr("abs:href");

       if(!linksList.contains(actualUrl) &
actualUrl.startsWith(baseURI)){
           print(" * a: <%s>  (%s)", actualUrl, trim(link.text(), 35));
           linksList.add(actualUrl);
           crawl(actualUrl);
       }
   }
}
```

El algoritmo obtiene el documento HTML de la url raiz (*rootURL)* y comienza por extraer todas las URLs que contiene, en formato absoluto por supuesto (de ahi la importancia de

guardar el baseURI junto a rootURI). En bucle, va accediendo a cada pagina y, de forma recursiva, vuelve a listar y acceder a las URLs contenidas en cada nivel. Para tener una visión más realista, es necesario conocer la clase entera:

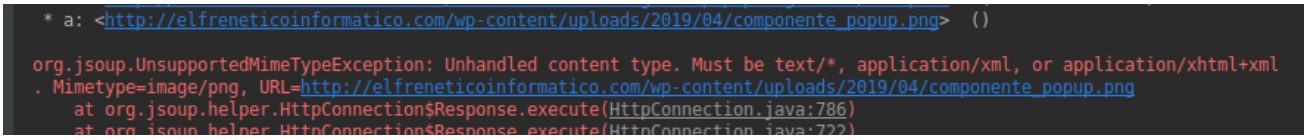
```
public class CrawlerEngine {

   private String baseURI;
   private String rootURI;
   private ArrayList<String> linksList;
   private Document doc;

   public CrawlerEngine(String baseURI) throws IOException {
       this.baseURI = baseURI;
       this.rootURI = String.valueOf(baseURI);
       linksList = new ArrayList<String>();
       doc = new DocumentLoader(baseURI).getDoc();
   }

   public CrawlerEngine(String baseURI, String rootURI) throws
IOException {
       this.baseURI = baseURI;
       this.rootURI = rootURI;
       linksList = new ArrayList<String>();
       doc = new DocumentLoader(rootURI).getDoc();
   }

   public void crawl() throws IOException {
       crawl(rootURI);
   }

   public void crawl(String url) throws IOException {
       SintaxEngine sintaxEngine = new SintaxEngine(url);
       Elements links = sintaxEngine.getAWithRef();

       for(Element link : links){
           String actualUrl = link.attr("abs:href");

           if(!linksList.contains(actualUrl) &
actualUrl.startsWith(baseURI)){
               print(" * a: <%s>  (%s)", actualUrl, trim(link.text(),
35));
```

```
                linksList.add(actualUrl);
                crawl(actualUrl);
            }
        }
    }

    public boolean linksEmpty(){
        return linksList.isEmpty();
    }

    private static void print(String msg, Object... args) {
        System.out.println(String.format(msg, args));
    }

    private static String trim(String s, int width) {
        if (s.length() > width)
            return s.substring(0, width-1) + ".";
        else
            return s;
    }
}
```

Clasificación del tipo de contenido

No obstante, en ciertos casos puede ocurrir un error inesperado respecto al tipo de contenido obtenido de una página web. Se ha mencionado antes que las páginas que contienen fuentes multimedia y demás no interesan, ya que no contienen información HTML que parsear. No obstante, es posible que, aún aplicando el filtro adecuado por selección de sintaxis, obtengamos enlaces a imágenes. Esto sucede cuando, en una web dada, las imágenes contienen un hipervínculo a su archivo fuente original, de forma que si el usuario hace click sobre la imagen será llevado hasta la página en la que aparece sólo la imagen. Es decir, la página de recurso.

Ante este problema, una solución universal es ignorar el tipo de contenido. Cuando analicemos el contenido HTML simplemente devolverá un objeto vacío, pero no es necesario que nos salte dicho error:



Para ignorar el tipo de contenido debe modificarse la conexión para evitar que compruebe el contenido.

```
public DocumentLoader(String url) throws IOException {
   this.doc = Jsoup.connect(url).ignoreContentType(true).get();
}
```

De esta forma, al ejecutar el algoritmo, es posible obtener todas las URLs del contenido de un sitio web dado.

Prueba del algoritmo recursivo

Se lleva a cabo un ejemplo con *elfreneticoinformatico.com*:

```
public class CrawlerEngineTest {

   private static final String URL =
"http://elfreneticoinformatico.com";
   private CrawlerEngine sut;

   @Before
   public void init() throws IOException {
       sut = new CrawlerEngine(URL);
   }

   @Test
   public void crawl() throws IOException {
       Date start = new Date();
       sut.crawl();
       Date finish = new Date();
       long mils = finish.getTime() - start.getTime();
       long seconds = mils / 1000;
       System.out.println("Total URLs crawled at " + URL + " :: " +
sut.getLinksList().size());
       System.out.println("Time :: " + seconds);
   }
```

```
}
```

Y el resultado (exitoso) muestra en pantalla una lista de todas las URLs scrapeadas y, al final, los dos mensajes de estadísticas indicados:

```
  * a: <http://elfreneticoinformatico.com/category/otros/page/2/>  (« Siguientes entradas)
  * a: <http://elfreneticoinformatico.com/category/otros/page/3/>  (« Siguientes entradas)
Total URLs crawled at http://elfreneticoinformatico.com :: 555
Time :: 135

Process finished with exit code 0
```

Patrón *Strategy*

Lo normal es que puedan configurarse diferentes estrategias de *crawling*. Para ello, se usará un patrón Strategy. Se genera una interfaz (llamada *Crawler*), con la siguiente estructura de ejemplo (variará a lo largo del desarrollo):

```
public interface Crawler {
   void crawl() throws IOException;

   ArrayList<String> getScrapedLinks();
}
```

Y, por supuesto, *CrawlerEngine* debe implementar dicha interfaz y sobreescribir los metodos obligatorios.

De esta forma, si se crea (que se creará) otra clase con una estrategia diferente de crawling, simplemente se inyectará como una implementación de la interfaz.

Optimización del algoritmo

El último ejemplo recursivo corresponde a un entorno conocido y conscientemente preparado para garantizar éxito. Además, se trata de un sitio pequeño, de poco más de 500 páginas. Existen algunos errores que se darán en espacios reales por diversos factores, que deben tenerse en cuenta y tratarse adecuadamente.

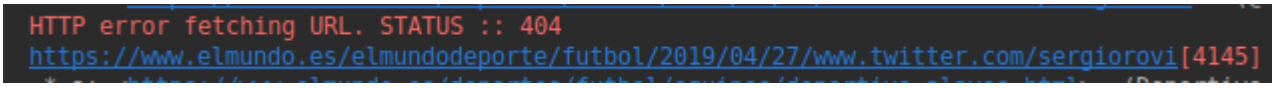
Para el tratamiento de errores, como en cualquier proyecto Java, se utilizará la sentencia *try{}catch*.

Errores de estado

Este código de error se da cuando la página a la que se intenta acceder no existe. Una de las propiedades a las que da acceso Jsoup es el código de estado (*statusCode*), y podemos gestionar todas las respuestas erróneas (que no sean del orden 20X) de una forma sencilla y universalizada:

```
} catch (HttpStatusException statusExcp){
   System.err.println(statusExcp.getMessage() + ". STATUS :: " +
statusExcp.getStatusCode());
   System.err.println(statusExcp.getUrl() + " [" + linksList.size() +
"]");
}
```

De esta forma, cuando no sea posible acceder a una página concreta, nos mostrará el mensaje de error (*error fetching URL)*, el código (como 404 o 503), la URL en cuestión y, entre corchetes, el tamaño de la lista de enlaces en ese momento:



El 404 es, probablemente, el que conlleva una solución más fácil. Simplemente, al obtener un código 404, no se inserta dicha dirección en la lista de links y se continúa con el bucle recursivo. No obstante, existen otros errores (como el 503 generalmente) cuya solución puede ser un dolor de cabeza.

Muchos sitios (como Amazon, por ejemplo) detectan y monitorizan los accesos a sus servidores. De forma que, cuando estamos recorriendo todas sus URLs de forma automatizada, cortan el servicio a esa IP sospechosa de robot. La solución más sencilla es generar un tiempo aleatorio de espera entre cada petición (entre 5 y 20 segundos, por ejemplo), de forma que nuestro bot se comporte de una forma más parecida a los humanos. No obstante, esta opción tiene una pega muy clara: limita mucho la cantidad de URLs obtenidas por unidad de tiempo.

Una solución que mejora el panorama respecto al tiempo de ejecución es activar un servicio de rotación de IPs. De esta forma, el servidor recibirá muchas peticiones, pero corresponderán a diferentes direcciones IP y, de esta forma, no detectará actividad sospechosa. Por un lado, esta solución solventa con ventaja el problema del tiempo de acceso

pero, evidentemente, necesita de una infraestructura mucho más mayor para su realización, y su complejidad aumenta considerablemente.

Tiempo de espera agotado

Por diversas circunstancias, como nuestra conexión a internet o cualquier problema en la comunicación cliente-servidor, puede que se agote el tiempo de espera al solicitar el contenido de una dirección concreta. En estos casos lo mejor es olvidar la URL que tarda demasiado y continuar buscando en el resto. Como recomendación, pueden guardarse las URLs cuyo timeout se ha excedido y, tras terminar de escalar los enlaces deseados, puede procederse a reintentar conectar con las direcciones problemáticas.

Esta extensión también puede ser tratada de forma individualizada y concreta:

```
} catch (SocketTimeoutException timeOutExcp){
   System.out.println(timeOutExcp.getMessage());
}
```

Fuera de rango (StackOverflow)

No. No se trata de publicidad del foro de preguntas sobre desarrollo más famoso del mundo.

Este problema se encontrará solo cuando se trate de escalar sitios muy grandes. No es tan descabellado como puede parecer: pensemos en elmundo.es (o cualquier diario de noticias), amazon, grupo AliBaba, etc.

Cuando se acumulan demasiadas URLs en la lista de URLs encontradas, puede que se de este error. Normalmente al superar cifras entorno a 6.000 enlaces es cuando se da este problema. También es posible tratarla como las demás:

```
} catch (StackOverflowError stackOverflowError){
   System.err.println(stackOverflowError.getMessage());
   System.err.println(stackOverflowError.getLocalizedMessage());
   System.err.println(stackOverflowError.toString());
   System.err.println("[" + linksList.size() + "]");
}
```

No obstante, dicho error no es tan preocupante como parece desde un principio. Para comprobar si realmente afecta o no a la hora de almacenar las URLs buscadas, se hará uso de una nueva clase para leer y escribir archivos en formato JSON, que se encuentra ubicada en el paquete "*filemanager*" de la aplicación. El esqueleto básico de dicha clase es:

```
public class JsonSerializer {

   private Object objectToSerialize;
   private Gson gson;

   public JsonSerializer(Object objectToSerialize){
       this.objectToSerialize = objectToSerialize;
       gson = new Gson();
   }

   public JsonSerializer(){
       gson = new Gson();
   }

   public void writeFile(String filename) throws IOException {
       try(FileWriter fileWriter = new FileWriter(filename)){
           gson.toJson(objectToSerialize, fileWriter);
           System.out.println("SERIALIZED TO :::::::::: " + filename);
           fileWriter.flush();
       } catch (Exception error){
           System.err.println("ERROR WRITING JSON FILE : " +
error.getMessage());
       }
   }

   public ArrayList<String> readJsonFile(String filename) throws
FileNotFoundException {
       JsonReader reader = new JsonReader(new FileReader(filename));
       TypeToken<ArrayList<String>> token = new
TypeToken<ArrayList<String>>() {};
       ArrayList<String> linksList = gson.fromJson(reader,
token.getType());
       return linksList;
   }

}
```

Haciendo uso de estos métodos, se modifica el algoritmo central del motor de *crawling* para que, cada 100 enlaces, escriba un fichero con el contenido de la lista de enlaces:

```
SintaxEngine sintaxEngine = new SintaxEngine(url);
Elements links = sintaxEngine.getAWithRef();

for(Element link : links){
   String actualUrl = link.attr("abs:href");

   if(!linksList.contains(actualUrl) & actualUrl.startsWith(baseURI)){
       print(" * a: <%s>  (%s)", actualUrl, trim(link.text(), 35));
       linksList.add(actualUrl);
       if((linksList.size() % 100) == 0){
           JsonSerializer serializer = new JsonSerializer(linksList);
           serializer.writeFile("links.json");
       }
       crawl(actualUrl);
   }
}
```

En este último extracto de código se ha omitido parte del método (el tratamiento de errores) para facilitar la lectura del ejemplo.

De la forma mostrada, cuando el tamaño de la lista de enlaces sea múltiplo de 100, se guardará todo su contenido en el archivo *links.json*.

Definición de la prueba

La prueba a realizar para conocer el nivel de gravedad del error consta de 3 pasos:

1) Ejecución de un crawler recursivo.

Utilizando el *CrawlerEngine,* sin límite de ejecuciones, para provocar a propósito el error mencionado. Se trata de una búsqueda muy larga, cercana (si puede ser) a las cien mil páginas scrapeadas, para comprobar, además, el comportamiento del algoritmo con grandes cantidades de enlaces analizados. Como se indicó anteriormente, cada 100 páginas añadidas a la lista, se escribirá el contenido en un archivo. Para la ejecución de este crawler se ha creado este test, marcado para ser ignorado y no afectar a los demás tests esenciales:

```
public class CrawlerEngineTest {

    private static final String URL = "https://www.elmundo.es/";
    private CrawlerEngine sut;

    @Before
    public void init() throws IOException {
        sut = new CrawlerEngine(URL);
    }

    @Ignore
    @Test
    public void infiniteCrawl() throws IOException {
        Date start = new Date();
        sut.crawl();
        Date finish = new Date();
        long mils = finish.getTime() - start.getTime();
        long seconds = mils / 1000;
        System.out.println("Total URLs crawled at " + URL + " :: " +
sut.getLinksList().size());
        System.out.println("Time :: " + seconds);
    }
}
```

2) Carga del archivo JSON con las URLs.

Para comprobar en qué estado han quedado las URLs visitadas, se ha elaborado otro test (también marcado como ignorado) con el que se lee el archivo, se muestra su contenido y, finalmente, se imprime el tamaño de la lista:

```
public class JsonSerializerTest {

    private JsonSerializer sut;

    @Before
    public void init(){
        sut = new JsonSerializer();
    }

    @Test
```

```
    public void writeFile() {
    }

    @Ignore
    @Test
    public void readJsonFile() throws FileNotFoundException {
        ArrayList<String> returned = sut.readJsonFile("links.json");
        for(String link : returned){
            System.out.println(link);
        }
        System.out.println("List size :: " + returned.size());
    }
}
```

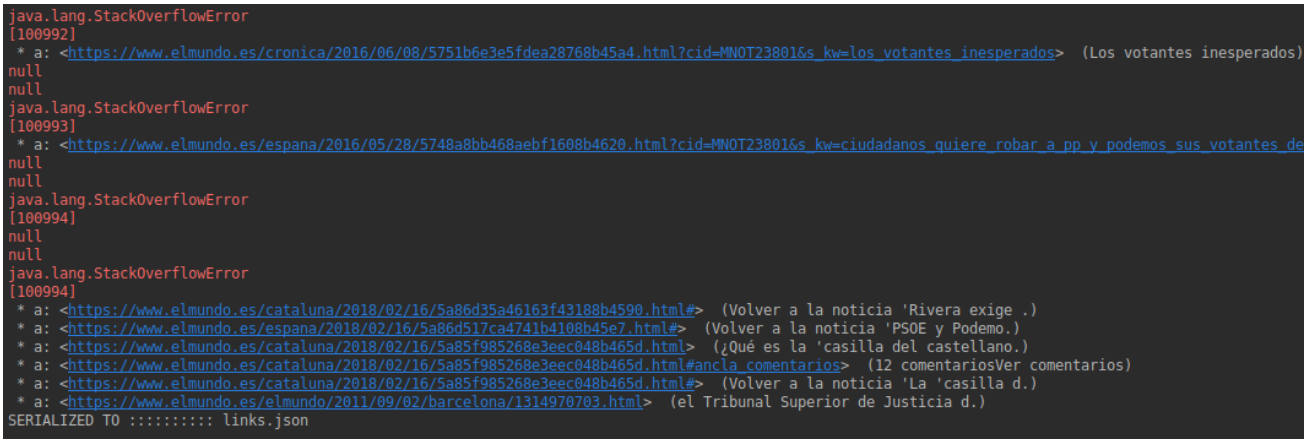
3) Comprobación de la integridad del archivo

Tras cargar y visualizar el conjunto de URLs guardadas en el archivo, deben comprobarse algunas consideraciones:

- El número de enlaces debe coincidir con la última llamada de escritura de la clase *JsonSerializer*. Si, por ejemplo, se han cargado 176 URLs, el archivo debe contener 100, ya que la llamada se hace cada 100 iteraciones.
- Deben verificarse como accesibles las URLs cuyo estado mostrado en la consola durante el crawler se ha mostrado con el error *stackOverflowError*. De forma que, si dichas URLs están en la lista, significa que el error no afecta al guardado de listas en el archivo.

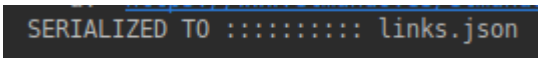
Ejecución

Tras varias horas de ejecución (aproximadamente algo menos de 4), se han obtenido 101.000 (ciento un mil) enlaces:



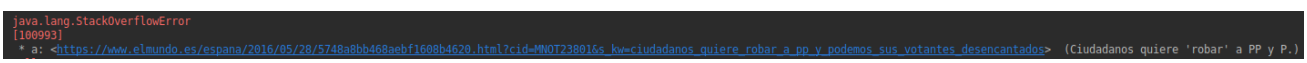
En la imagen puede verse, en rojo, diferentes impresiones en pantalla del error buscado: *java.lang.StackOverflowError*. Pero, ¿Ha alterado este error el algoritmo?

Abajo, en la imagen, puede verse el log de serializado:



Si abrimos el archivo con los datos en formato json desde un explorador (Mozilla Firefox, se demora varios segundos en abrir) y navegamos hasta el final del documento, observamos lo siguiente:

En total, se observan 100999 elementos en la lista. Es decir, 101000 (empieza en cero). Por tanto, primera prueba de que el algoritmo funciona bien aunque lance mensaje de stackOverflowError. Volviendo a la imagen anterior del log en consola, vemos que el enlace 100993 es el siguiente:



Para evitar confusiones con la imagen, el enlace exacto es:

- https://www.elmundo.es/espana/2016/05/28/5748a8bb468aebf1608b4620.html?cid=MNOT23801&s_kw=ciudadanos_quiere_robar_a_pp_y_podemos_sus_votantes_desencantados

Si volvemos a la imagen del navegador, observamos que el elemento número 100992 (uno menos, recordemos que empieza contando en cero) es exactamente el mismo:

- 100992: "https://www.elmundo.es/cronica/2016/06/08/5751b6e3e5fdea28768b45a4.html?cid=MNOT23801&s_kw=los_votantes_inesperados"

Y, si visitamos dicho enlace, podemos ver perfectamente dicha noticia, fechada en 8 de junio de 2016.

Además, si cargamos el archivo con el método del test creado anteriormente para este propósito, se imprimen todos los links y se muestra su tamaño en pantalla, que coincide con el mostrado por el navegador.

Tras esta prueba, se realiza una mucho más exhaustiva de entorno a 15 horas de duración. Se obtienen 250000 (dosciento cincuenta mil) enlaces y se realiza el mismo esquema de prueba descrito anteriormente, resultando exitosa.

Conclusiones

Tras la realización de las pruebas, se concluye que el *StackOverflowError*, si se trata de una manera adecuada, **no afecta al funcionamiento del algoritmo**.

Si no se trata, la aplicación crashea durante la ejecución y se detiene, como cualquier excepción no tratada.

Para tratarla adecuadamente, simplemente puede imprimirse un mensaje en pantalla y, de esta forma, se evita que el lanzamiento de dicha excepción provoque la caída del sistema.

Evitar anti-bots

Un problema que sí produce verdaderos dolores de cabeza durante la exploración de un sitio web viene dado por las defensas de los servidores frente a bots de este tipo. Cuando, de forma automatizada, estás haciendo muchas llamadas a diferentes URLs de un servidor desde

la misma dirección IP, el sitio puede detectarlo como un bot y bloquear el servicio temporalmente a dicha dirección. De forma que, para entonces, el crawler ya no sirve para nada, ya que solo devolverá errores (generalmente del tipo 503).

Para evitar este error, la solución más sencilla es esperar un tiempo. Se puede implementar de una forma muy sencilla:

```
Thread.sleep(5 * 1000);
```

De esta forma, el proceso esperará 5 segundos cuando le sea indicado. Combinándolo con un algoritmo generador de números aleatorios, puede configurarse un tiempo aleatorio de espera, lo que reduce en mucho las posibilidades de ser cazados como robots.

Tras la modificación del algoritmo, ya se obtienen dos estrategias de scrapeado:

1) **Estrategia recursiva simple:** sin tiempo de espera. Análisis de un sitio completo utilizando recursividad para acceder a todas las URLs posibles.
2) **Estrategia recursiva con tiempo de espera:** igual que la anterior pero, esta vez, introduciendo un tiempo de espera aleatorio para evitar ser detectados como bots.

La clase *SlicedCrawlerEngine* contendrá el nuevo algoritmo, y heredará los atributos y métodos base de *CrawlerEngine*:

```
public class SlicedCrawlerEngine extends CrawlerEngine{

   public SlicedCrawlerEngine(String baseURI) throws IOException {
       super(baseURI);
   }

   public SlicedCrawlerEngine(String baseURI, String rootURI) throws
IOException {
       super(baseURI, rootURI);
   }

   @Override
   public void crawl(String url){
       try{
           SintaxEngine sintaxEngine = new SintaxEngine(url);
           Elements links = sintaxEngine.getAWithRef();
```

```
            for(Element link : links){
                String actualUrl = link.attr("abs:href");

                if(!super.linksList.contains(actualUrl) &
actualUrl.startsWith(super.baseURI)){
                    super.print(" * a: <%s>  (%s)", actualUrl,
super.trim(link.text(), 35));
                    linksList.add(actualUrl);
                    if((linksList.size() % 100) == 0){
                        JsonSerializer serializer = new
JsonSerializer(linksList);
                        serializer.writeFile("links.json");
                    }
                    Thread.sleep(randomTime());
                    crawl(actualUrl);
                }
            }
        } catch (HttpStatusException statusExcp){
            System.err.println(statusExcp.getMessage() + ". STATUS :: " +
statusExcp.getStatusCode());
            System.err.println(statusExcp.getUrl() + " [" +
linksList.size() + "]");
        } catch (SocketTimeoutException timeOutExcp){
            System.out.println(timeOutExcp.getMessage());
        } catch (Exception error){
            System.out.println(error.getMessage());
        } catch (StackOverflowError stackOverflowError){
            System.err.println(stackOverflowError.toString());
            System.err.println("[" + linksList.size() + "]");
        }
    }

    private int randomTime(){
        Random random = new Random();
        return (random.nextInt(5) + 1) * 1000;
    }
}
```

Y, similar a su padre, el nuevo motor debe ser testeado:

```
public class SlicedCrawlerEngineTest {

    private static final String URL = "https://www.elmundo.es";
    private SlicedCrawlerEngine sut;

    @Before
    public void init() throws IOException {
        sut = new SlicedCrawlerEngine(URL);
    }

    @Ignore
    @Test
    public void infiniteSlicedCrawl() throws IOException {
        Date start = new Date();
        sut.crawl();
        Date finish = new Date();
        long mils = finish.getTime() - start.getTime();
        long seconds = mils / 1000;
        System.out.println("Total URLs crawled at " + URL + " :: " +
sut.getLinksList().size());
        System.out.println("Time :: " + seconds);
    }
}
```

Búsquedas

Con un crawler como el anterior se pueden obtener innumerables enlaces a documentos web, pero es necesario poder buscar contenido en ellos.

Para representar este caso se implementará un buscador que verificará si las páginas de una lista dada contienen, en un párrafo (<p>) una palabra específica.

El algoritmo del motor de búsqueda:

```
public class SearchEngine {

   private JsonSerializer serializer;
   private ArrayList<String> matches;

   public SearchEngine(){
       serializer = new JsonSerializer();
       matches = new ArrayList<>();
   }

   public ArrayList<String> search(String fileName, String target)
throws IOException {
       ArrayList<String> links = serializer.readJsonFile(fileName);
       for(String link : links){
           Element body = new DocumentLoader(link).getBody();
           if(!body.getElementsContainingText(target).isEmpty()){
               matches.add(link);
           }
       }
       return matches;
   }
}
```

De esta forma es posible seleccionar aquellas páginas que contengan una palabra o texto especificado. Y, por supuesto, no pueden faltar los tests:

```
public class SearchEngineTest {

   private static final String URI = "http://books.toscrape.com/";
```

```
    private static final String TARGET = "united states";
    private static final String FILE_TO_READ =
"src/main/resources/links.json";
    private static final String FILE_TO_WRITE =
"src/test/resources/matches.json";

    private CrawlerEngine crawler;
    private SearchEngine sut;

    @Before
    public void init() throws IOException {
        sut = new SearchEngine();
        crawler = new CrawlerEngine(URI);
    }

    @Ignore
    @Test
    public void crawl_and_search() throws IOException {
        crawler.crawl();
        ArrayList<String> result = sut.search(crawler.getJsonFileName(),
TARGET);
    }

    @Ignore
    @Test
    public void only_search() throws IOException {
        ArrayList<String> result = sut.search(crawler.getJsonFileName(),
TARGET);
        for(String l : result){
            System.out.println(l);
        }
    }
}
```

A continuación, se muestra una captura de uno de los resultados obtenidos:

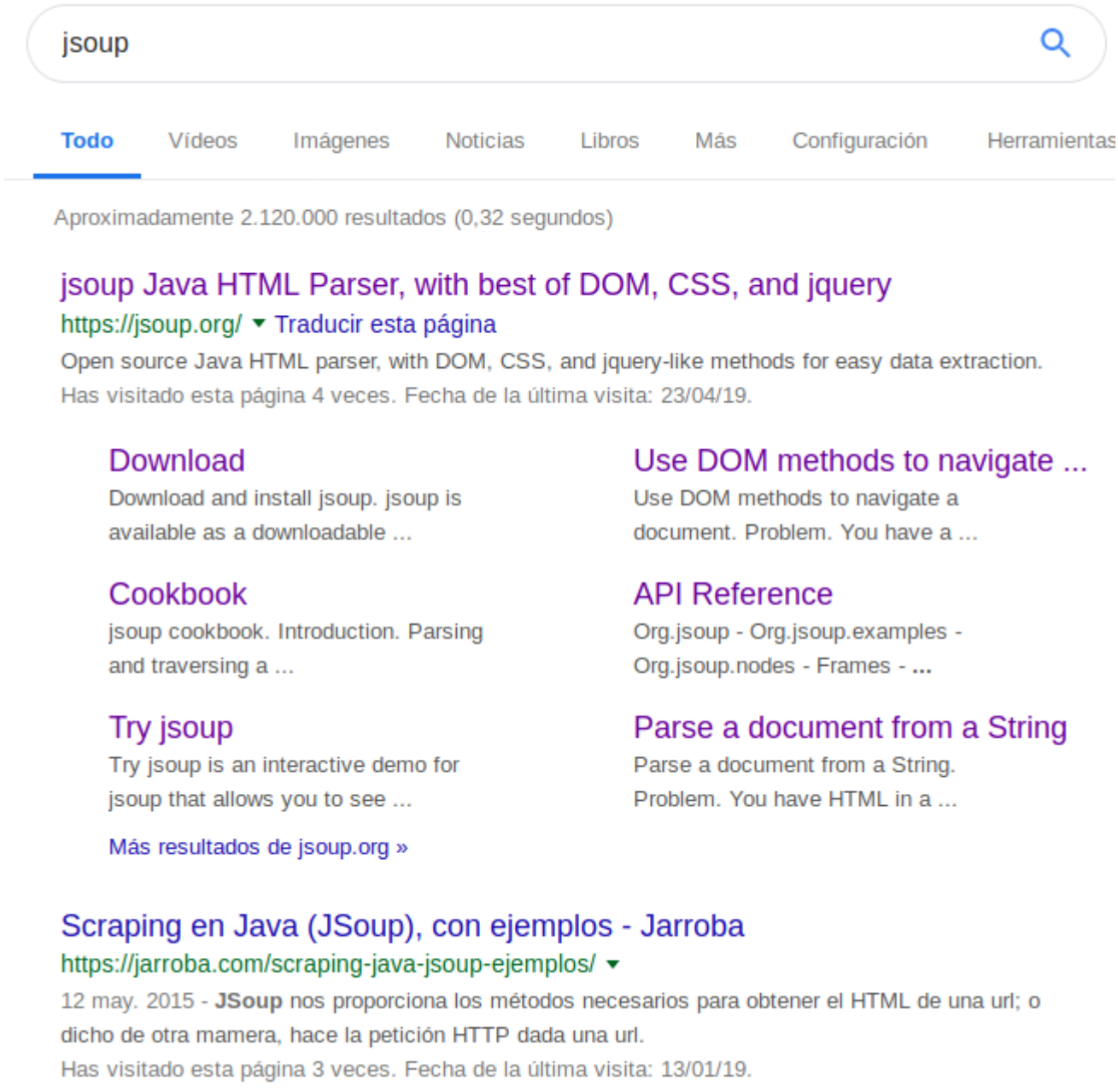
Product Description

Shot down over Siberia in what was to be a simple meet-and-greet-mission, ex-Justice Department age
loyalty to the former Soviet Union has festered for decades into an intense hatred of the United States.E

Patrón google

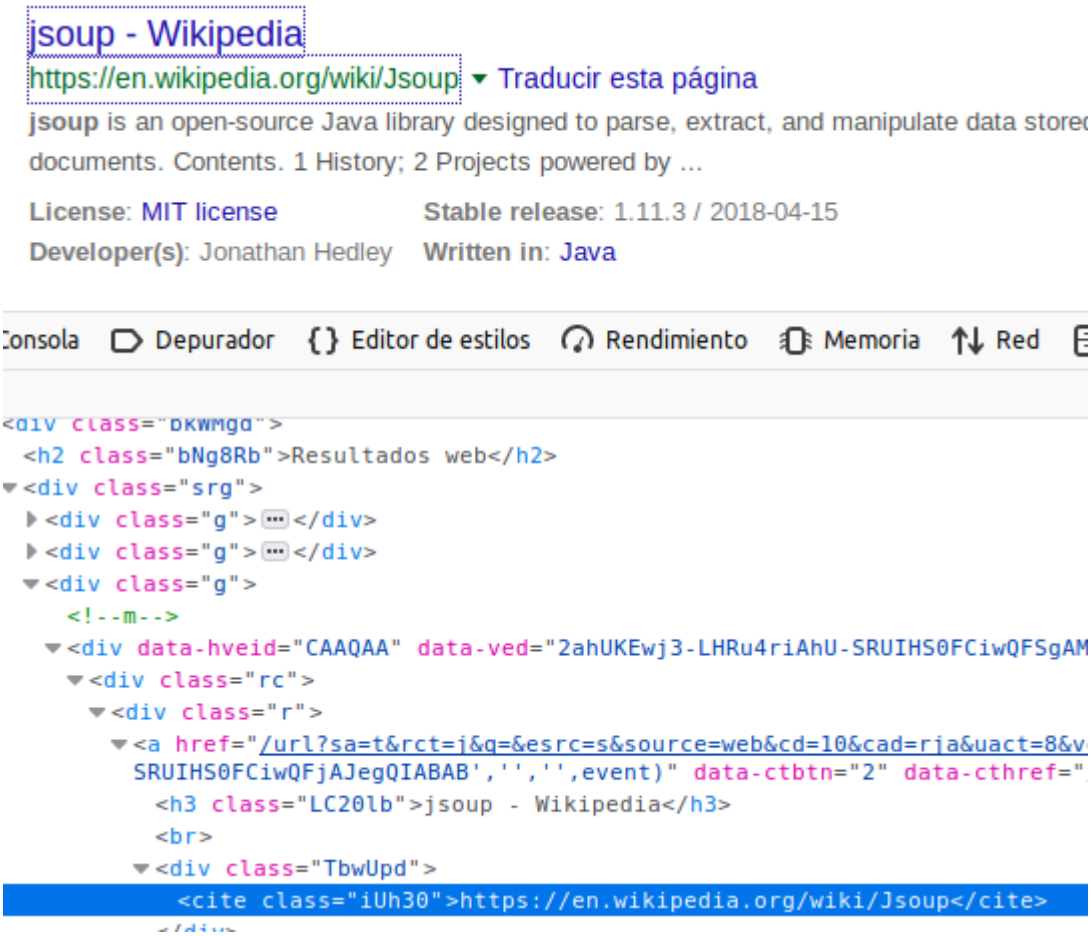
Hasta ahora, el crawler recoge cualquier URL encontrada en la página y, de forma recursiva, va accediendo a todos los documentos del sitio. En ciertos casos, esto es útil, pero no siempre.

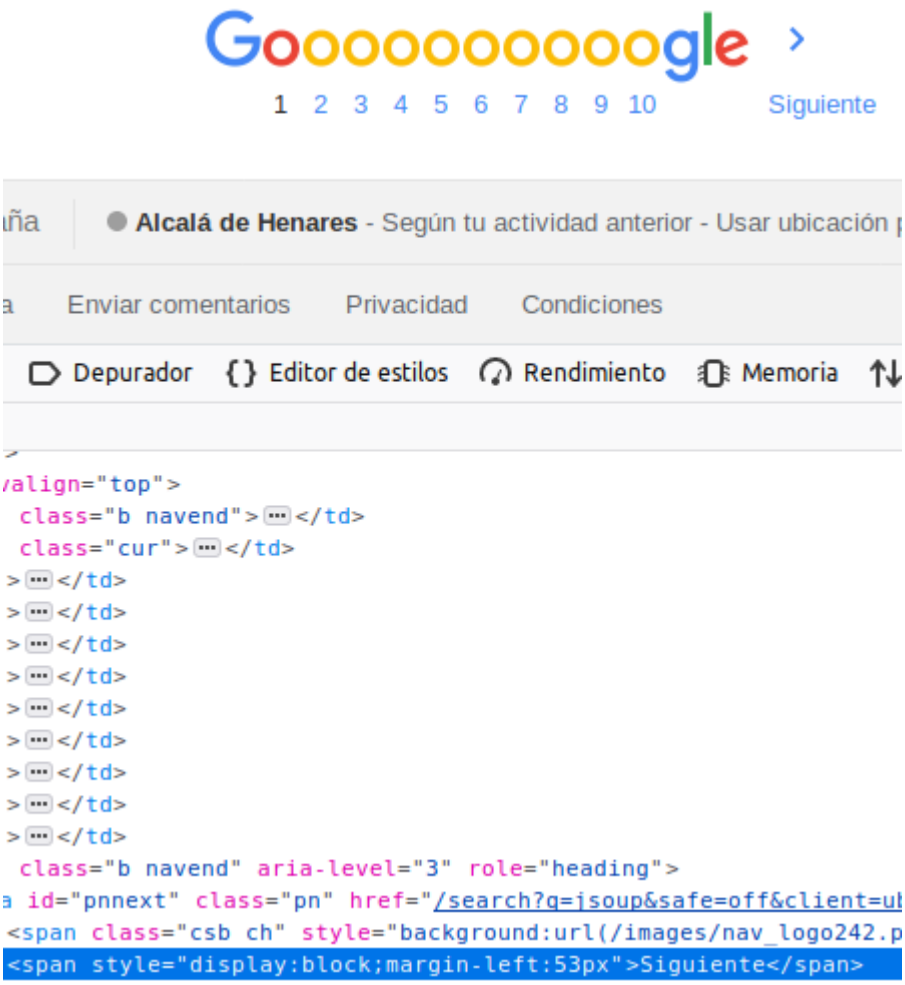
Pensemos en los resultados que arroja el buscador Google:



Hay algunos enlaces, como los de la barra superior, que no son interesantes. Sin embargo, nos interesan los enlaces a los que llevan los títulos de cada resultado. Y además, al final de la página, tenemos una forma muy sencilla de acceder a la siguiente página de resultados:

Con la ayuda de un navegador común (Firefox), no es difícil obtener las clases que buscamos en todo este conjunto de información:





Por tanto, este patrón de búsqueda consta de dos elementos:

- Clase para los enlaces
- Clase para avanzar a la siguiente página

En lugar de hacer un bucle recursivo, solo debe avanzar en cada página por los elementos deseados y, al terminar, ir al elemento marcado como siguiente y realizar la misma búsqueda en la siguiente página, hasta alcanzar el límite de páginas dado por el usuario, o que no queden más páginas.

Motor de búsqueda Patrón Google:

```
public class GooglePatternEngine implements Crawler{

   private static final int FIRST_PAGE_NUMBER = 1;
   private static final String DEAFULT_JSON_FILE_NAME =
"src/main/resources/links_google.json";
```

```
    private ArrayList<String> linksList;
    private String actualPage;
    private String tag;
    private String nextPageTag;
    private int pageLimit;
    private String jsonFileName;

    public GooglePatternEngine(String parentUrl, String tag, String
nextPageTag, int pageLimit){
        this.actualPage = parentUrl;
        this.tag = tag;
        this.nextPageTag = nextPageTag;
        this.linksList = new ArrayList<>();
        this.pageLimit = pageLimit;
        jsonFileName = DEAFULT_JSON_FILE_NAME;
    }

    @Override
    public void crawl() throws IOException {
        parsePage(actualPage, FIRST_PAGE_NUMBER);
        saveLinksList();
    }

    public void parsePage(String pageUrl, int pageNum) throws IOException
{
        System.out.println("PAGE :: " + pageNum);
        Document document = new DocumentLoader(pageUrl).getDoc();
        Elements elements = document.select(tag);
        for(Element element : elements){
            linksList.add(element.text());
            System.out.println(element.text());
        }
        if(pageNum < pageLimit &&
!document.select(nextPageTag).attr("abs:href").isEmpty()){
            pageNum ++;
            parsePage(document.select(nextPageTag).attr("abs:href"),
pageNum);
        } else{
            System.out.println("Finished");
        }
```

```
    }

    @Override
    public ArrayList<String> getLinksList() {
        return linksList;
    }

    private void saveLinksList() throws IOException {
        JsonSerializer serializer = new JsonSerializer();
        serializer.writeFile(linksList, jsonFileName);
    }
}
```

Y, con un test muy sencillo, se verifica su funcionamiento:

```
public class GooglePatternEngineTest {

    private static final String PARENT_URI =
"https://www.google.com/search?client=ubuntu&channel=fs&q=jsoup&ie=utf-
8&oe=utf-8";
    private static final String TAG = "cite.iUh30";
    private static final String NEXT_PAGE_TAG ="a#pnnext";
    private static final int PAGE_LIMIT = 5;
    private GooglePatternEngine sut;

    @Before
    public void init(){
        sut = new GooglePatternEngine(PARENT_URI, TAG, NEXT_PAGE_TAG,
PAGE_LIMIT);
    }

    @Test
    public void crawl() throws IOException {
        sut.crawl();
    }
}
```

Patrón e-commerce

Dándole una pequeña vuelta de tuerca más al patrón Google, puede deducirse que escanear solo la lista de enlaces, al fin y al cabo, no produce diferencias respecto a una búsqueda recursiva en el resultado, ya que solo obtiene enlaces brutos.

Un caso en el que puede resultar interesante obtener más información de cada página visitada, puede darse al analizar un portal de comercio electrónico. El ejemplo se va a basar en el portál *books.toscrape.com*, creado especialmente para probar este tipo de algoritmos.

Al igual que el patrón Google, dispone al final de la página de elementos de navegación, lo que permitirá avanzar con libertad entre las diferentes páginas.

Los elementos interesantes para este ejemplo son:

- Enlace al libro
- Título
- Precio

Al igual que en el patrón anterior, avanzará por cada elemento de la página obteniendo el enlace a los productos. Con dicho enlace, accederá a la página del producto y obtendrá el resto de datos deseados.

Con este patrón se consigue una optimización de la búsqueda para sitios de venta online, basado en parte en el patrón google.

Modo de empleo

Tras todo el desarrollo mencionado en el último apartado, se debe tener en cuenta qué pasos seguir para utilizar todo este software embebido en cualquier otro tipo de sistema.

Obtención de enlaces (crawler)

Lo primero, y más importante, es utilizar uno de los diversos patrones creados para explorar todo el contenido de un sitio web concreto y obtener todos los enlaces posibles.

Análisis (motor de búsqueda)

Con el buscador implementado, se filtran los enlaces y se extraen aquellos de verdadero interés para la finalidad deseada.

Exportación

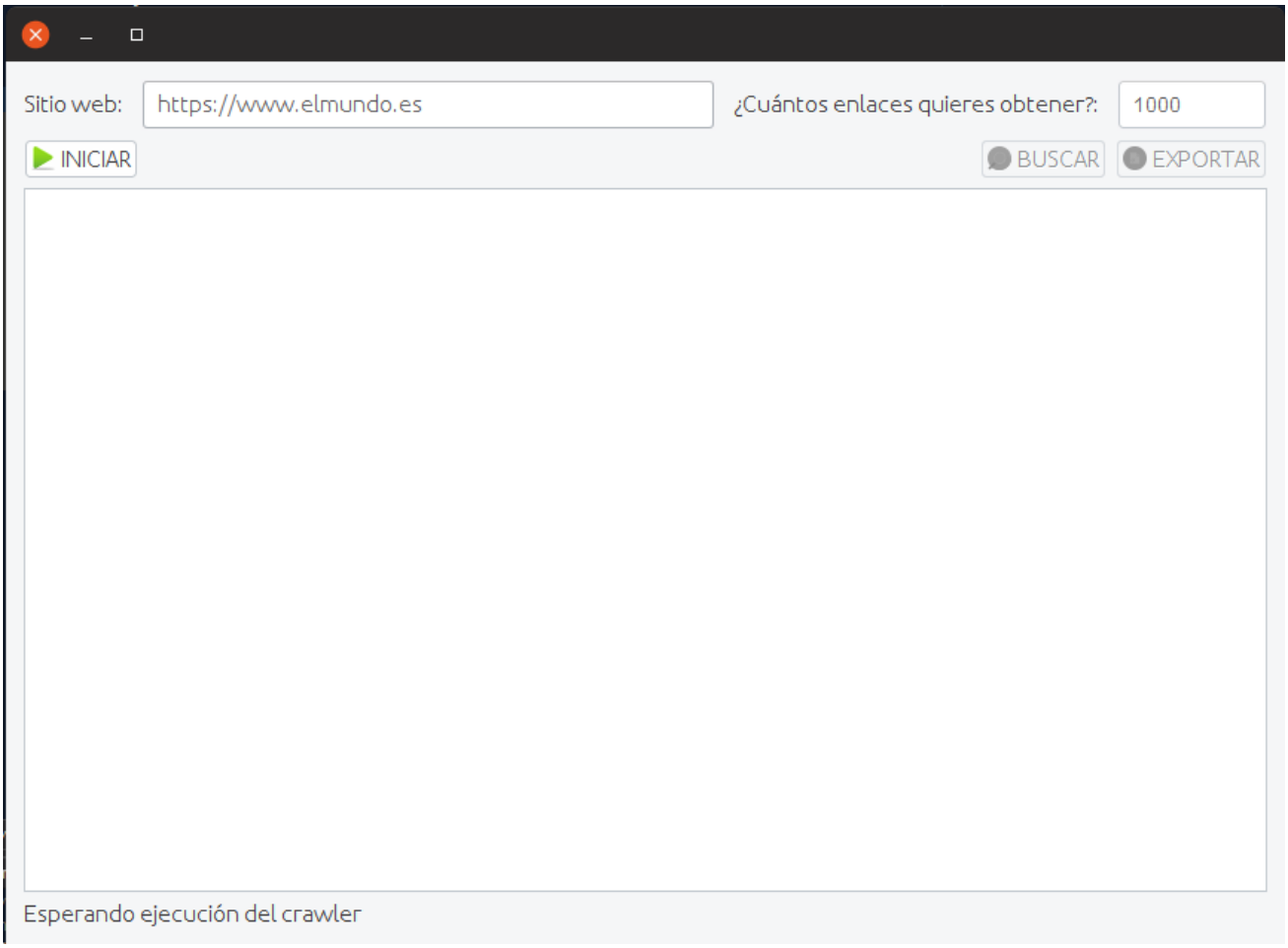
En el caso de este software, se ha implementado la exportación del contenido en formato JSON a un archivo de texto. No obstante, de manera sencilla gracias al patrón strategy, pueden crearse nuevas implementaciones que permitan mandar esa información a través de una API o lo que el desarrollador desee.

Prueba con entorno gráfico

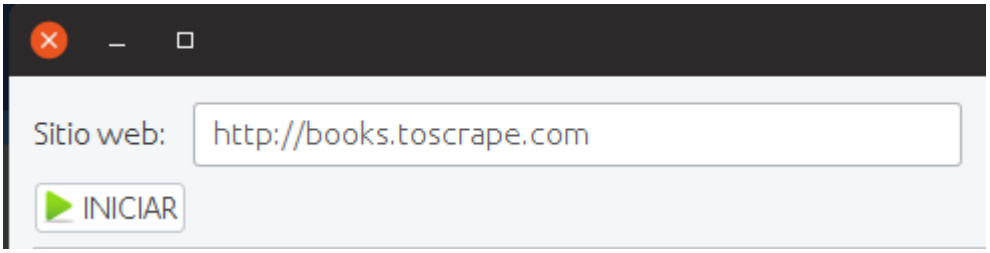
Para probar la modularidad de la librería creada, se ha creado una aplicación Swing con funcionalidad básica para obtener enlaces de cualquier web. Dicha aplicación implementa el algoritmo recursivo explicado en apartados anteriores.

Manual de usuario

Al arrancar la aplicación, aparece la ventana principal:

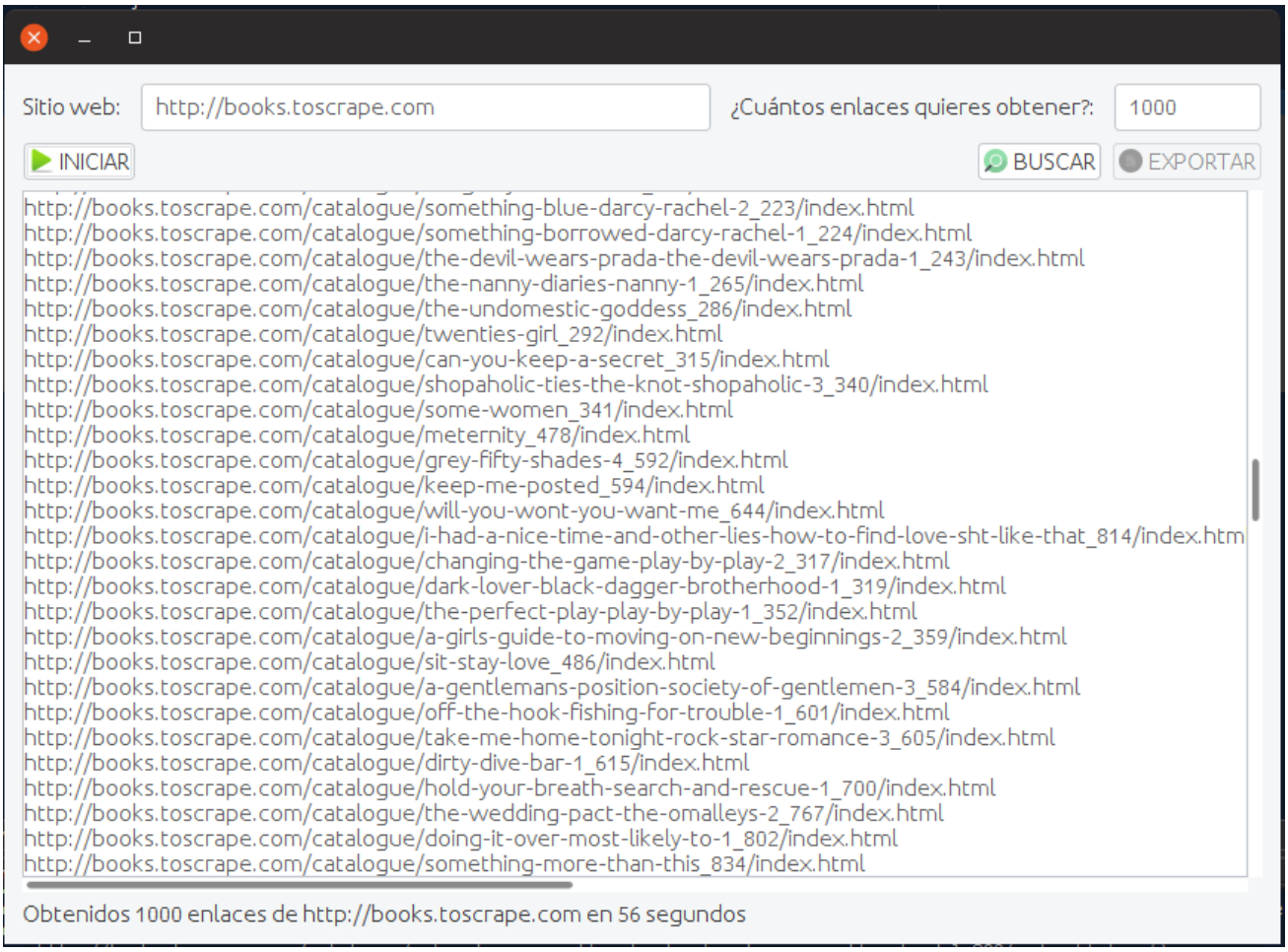


Por defecto, aparece la web de elmundo.es. Puede introducirse la que el usuario desee, por ejemplo:



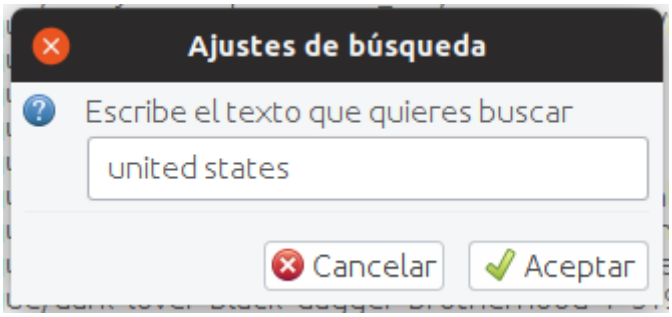
Los botones "buscar" y "exportar" están deshabilitados. El botón "buscar" se habilita cuando existen enlaces guardados tras ejecutar el crawler, de forma que el usuario puede buscar qué enlaces contienen la información relevante. A la derecha, arriba, el usuario también puede modificar qué cantidad de enlaces desea obtener (para evitar ejecuciones infinitas).

Al terminar la ejecución del crawler se muestran los enlaces recuperados y estadísticas del tiempo tardado en obtenerse:

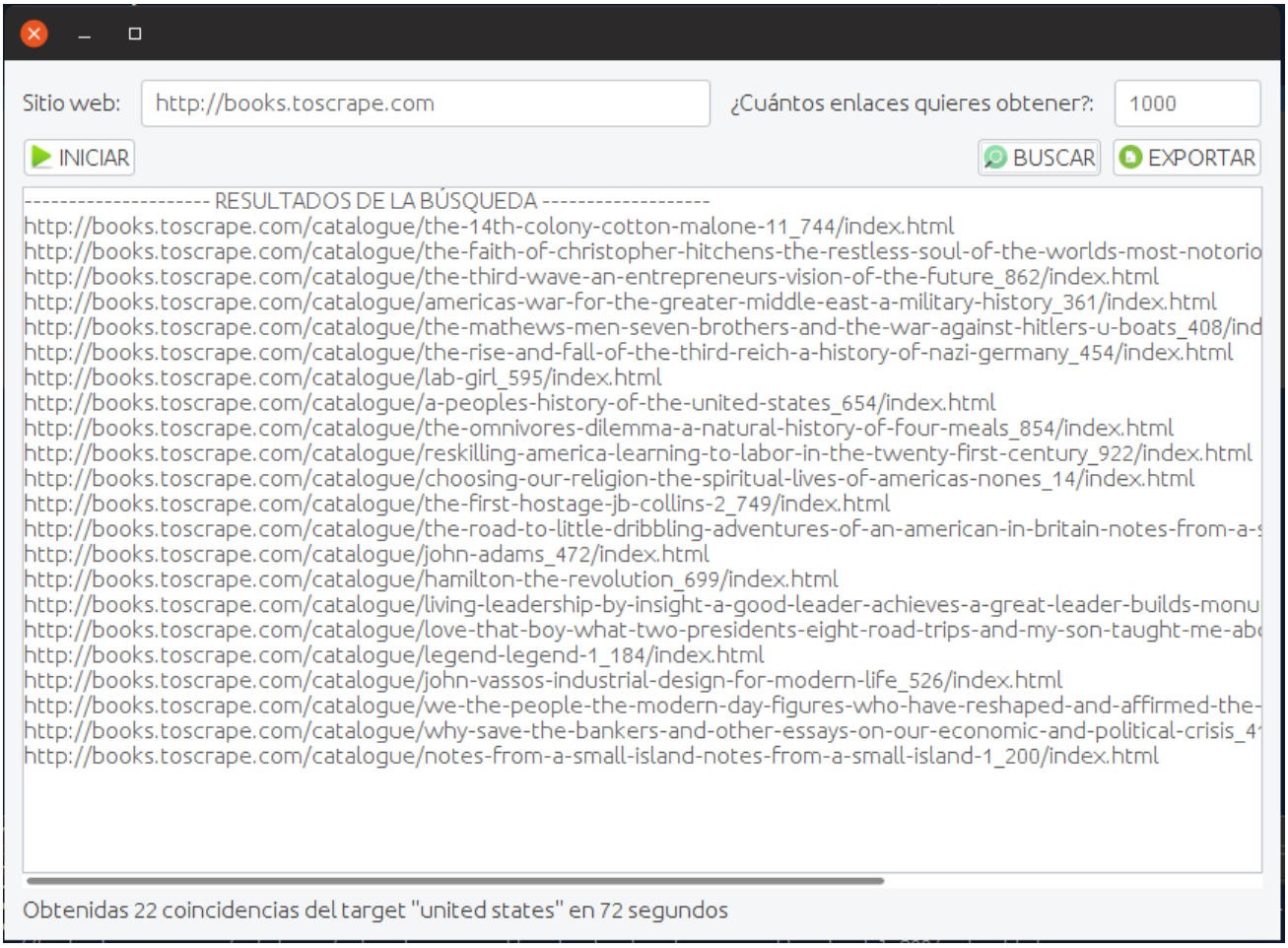


Como puede observarse, ha tardado 56 segundos en obtener mil enlaces. Esta estadística dependerá de lo lenta o rápida que sea, por un lado, la conexión a internet del cliente y, por otro, el tiempo de respuesta del servidor.

Como puede observarse en la última captura, ya está habilitado el botón de buscar. Si se pulsa, aparece un cuadro de diálogo preguntando al usuario qué desea buscar:

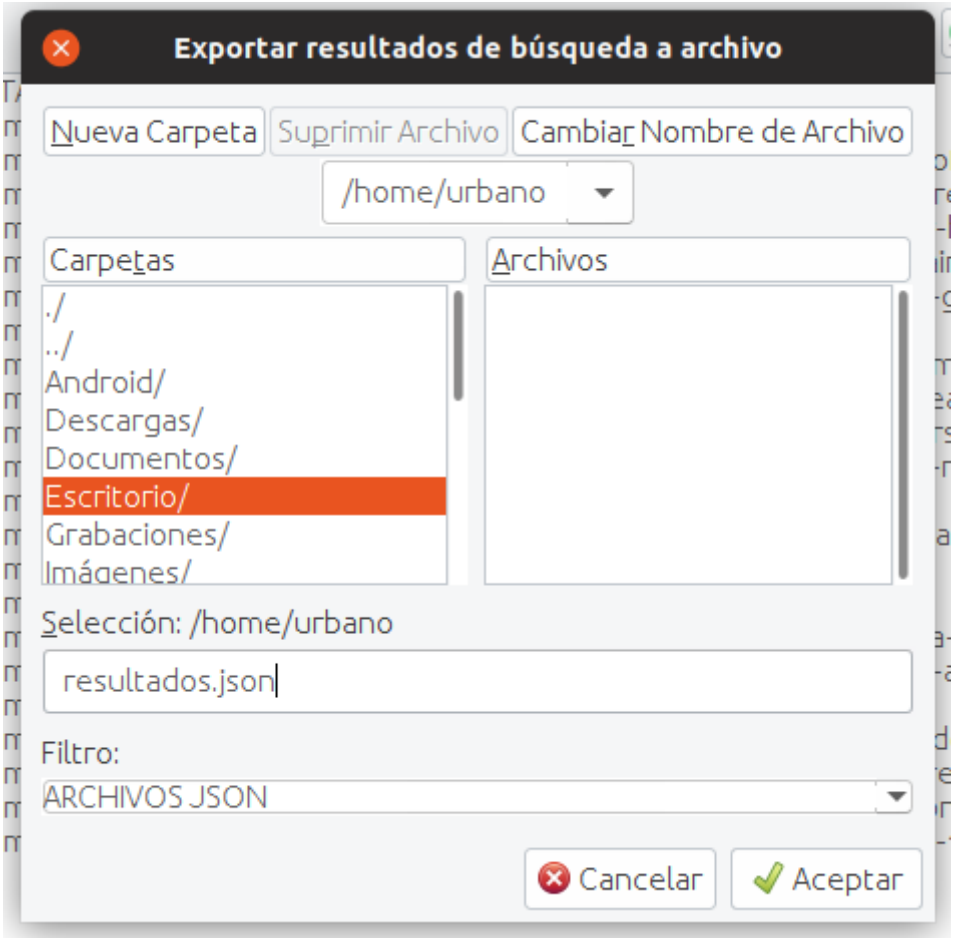


En el ejemplo se buscan las palabras "united states". Al pulsar "aceptar" el motor de búsqueda accederá a las mil URLs obtenidas en el paso anterior, analizará su contenido y, si contienen las palabras buscadas, las añadirá a la lista de elementos encontrados, mostrando los resultados en pantalla:



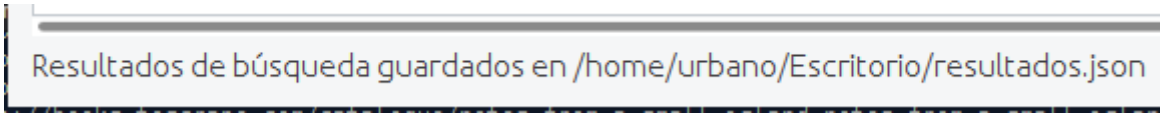
Ha tardado 72 segundos, lo que significa que, en este caso, la conexión ha ido más lenta. Ahora, que ya existen elementos coincidentes con la búsqueda, el botón de exportar ya está

habilitado. Lo que hace el botón es guardar el resultado en el archivo que el usuario desee. Al pulsar dicho botón, el sistema muestra una pantalla de navegación de archivos para que el usuario guarde el archivo donde desee, así como el nombre también queda a disposición del usuario. El formato JSON es el preferible.

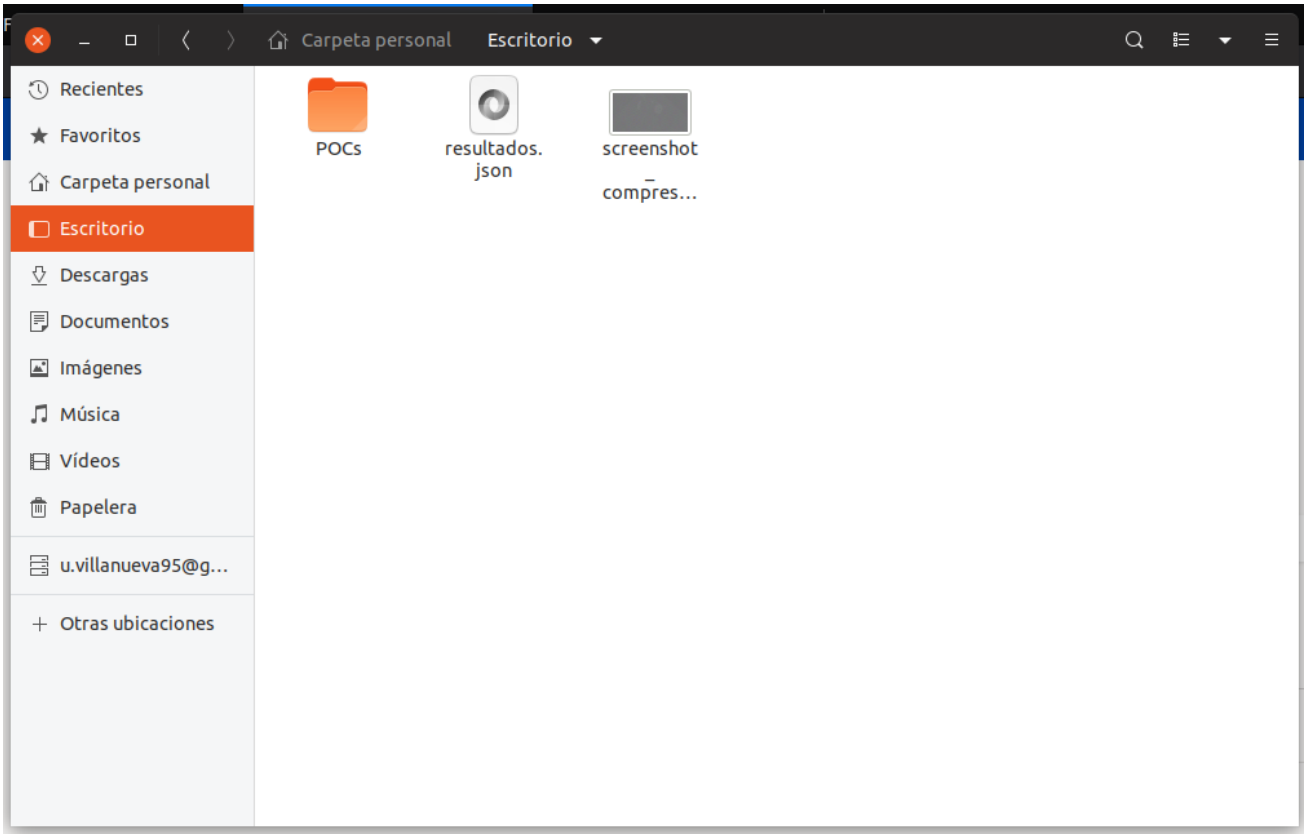


Se desea guardar el archivo en el escritorio con el nombre que se muestra en pantalla.

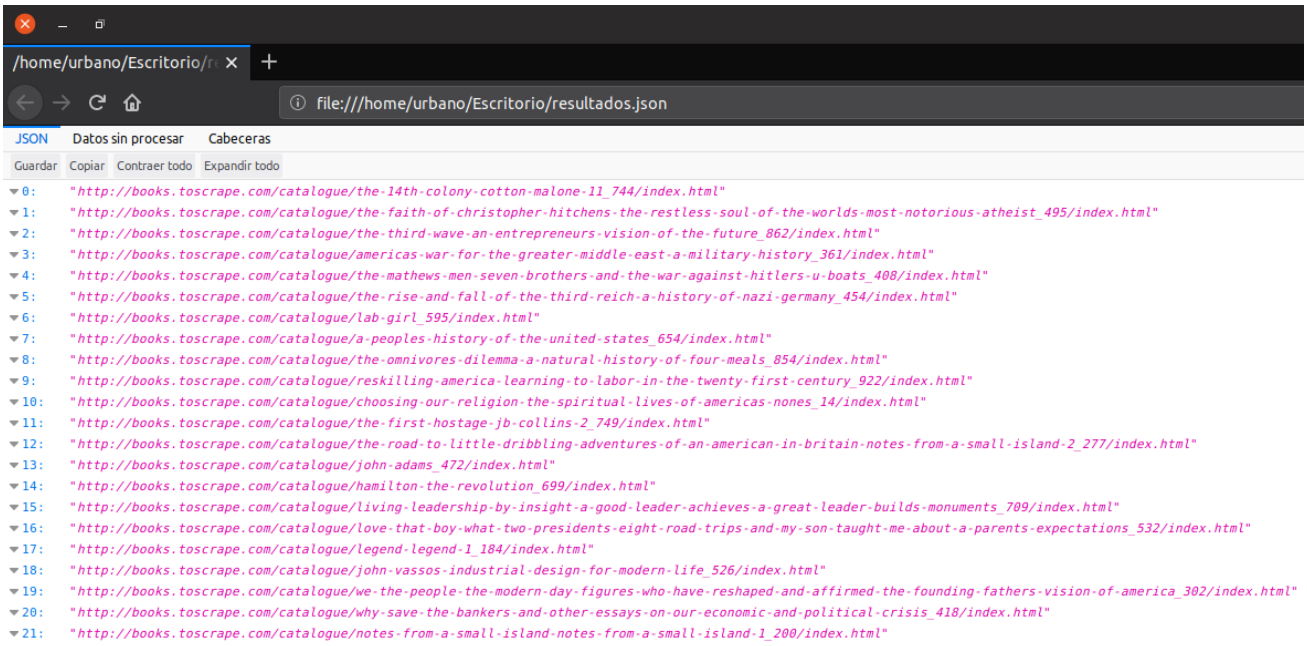
Al pulsar aceptar o presionar intro, la aplicación verifica que se ha guardado:



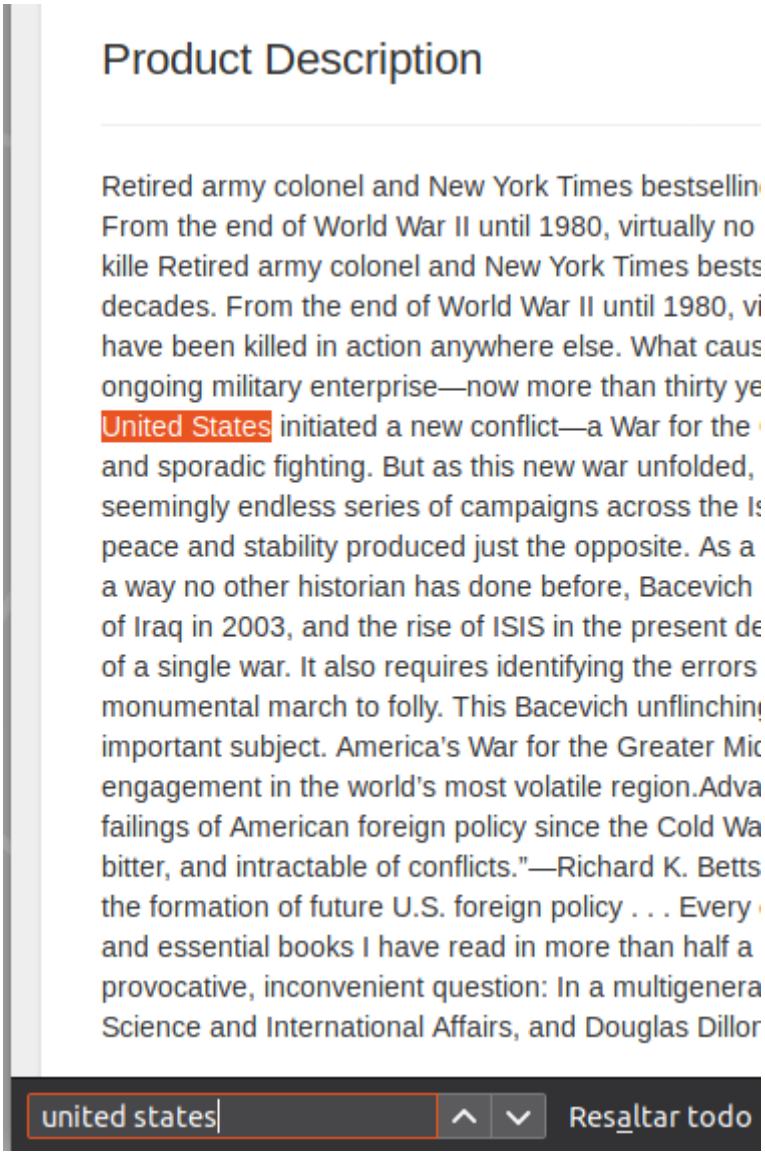
Y, al mostrar la carpeta Escritorio, se ve el archivo guardado:



Abrimos el archivo JSON con el navegador Firefox:



Si pulsamos en cualquier enlace redirigirá al usuario a la página en cuestión y puede comprobarse que dichas páginas contienen el target deseado. Para facilitar la búsqueda se recomienda utilizar el buscador embebido en el navegador (CTRL + F):



En esta prueba de concepto con entorno gráfico se visualizan claramente los 3 pasos que suceden durante el proceso:

- Scraping de enlaces
- Búsqueda personalizada
- Tratamiento de resultados, generando archivo JSON

Conclusiones

El web scraping no es, en sí, sólo un software. Es mucho más allá de una simple herramienta. La personalización y la posibilidad de implementar nueva funcionalidad son claves en un mercado cada vez más exigente. Se ha visto cómo las librerías más potentes para elaborar

bots no tienen valor (de hecho, son gratuitas), mientras que las empresas más caras son aquellas que separan totalmente la capa de recolección de datos del servicio ofrecido: Data As A Service.

Utilizando patrones de diseño se han elaborado los cimientos de lo que podemos definir como un buen crawler, que permite seccionar targets estandarizando diferentes tipos de motores para obtener información según qué tipo de página y, además, de una forma sencilla permite modificar y ampliar la funcionalidad respetando los principios básicos del buen software, obteniendo, por tanto, un concepto de sistema modulable, escalable, personalizable y estandarizado.

Además, se ha definido un método de 3 pasos que permite:

- Obtener URLs personalizando el algoritmo
- Buscar y filtrar los resultados según el contenido deseado
- Tratamiento final del resultado y exportación a formatos estandarizados comunes

Entorno y herramientas

A continuación se listan todas las herramientas utilizadas para el desarrollo de la obra. Esto incluye software, librerías importadas, entornos de desarrollo, etc:

- **Ubuntu:** versión 19.04
- **Git:** versión 2.20.1
- **Intellij Idea:** ultimate, licencia de la UAH para el autor de la obra, válida hasta enero de 2020. Versión 2019.01
- **NetBeans:** versión 8.2
- **Zotero:** gestión bibliográfica. Versión 5.0.66. Incluye extensión para navegador Firefox y complemento para Google Docs.
- **Docker:** herramienta para virtualización, utilizada en pruebas de concepto para persistencia de datos. Versión 18.09.5
- **Kitematic:** gestor gŕafico de Docker
- **Debeaver community:** 6.0.3
- **Mysql:** dockerizada. Versión 8.0
- **JDK:** 1.8.211
- **Jsoup:** 1.11.3 (importado con maven)
- **Junit:** 4.12 (importado con maven)
- **Hamcrest:** all, 1.3 (importado con maven)
- **Swing:** para prueba entorno gráfico
- **Gson:** 2.8.5 (importado con maven)
- **Mozilla Firefox:** 66.0.3 64bit
- **Google Docs:** software online para elaboración de documento escrito

Otras obras del autor

Durante la realización de esta obra, y utilizando los conocimientos adquiridos, el autor ha publicado un artículo sobre la misma temática. Dicho artículo será citado a continuación para demostrar la autoría y verificar que se trata del mismo autor, pudiendo coincidir código fuente, terminología y texto con esta obra:

U. V. Rodríguez, "Implementando un crawler sencillo con Jsoup," *Adictos al trabajo*, 14-May-2019. [Online]. Available: https://www.adictosaltrabajo.com/2019/05/14/implementando-un-crawler-sencillo-con-jsoup/. .

Bibliografía

[1]

"java - Is jsoup Document thread safe?," *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/46407118/is-jsoup-document-thread-safe?r=SearchResults&s=2|72.0702. [Accessed: 17-May-2019].

[2]

baeldung, "StackOverflowError," *Baeldung*, 15-May-2017. [Online]. Available: https://www.baeldung.com/java-stack-overflow-error. [Accessed: 06-May-2019].

[3]

"Jsoup Java Html Parser Tutorial," *o7planning.org*. [Online]. Available: https://o7planning.org/en/10399/jsoup-java-html-parser-tutorial. [Accessed: 29-Apr-2019].

[4]

"How To Monitor Docker Containers On Ubuntu Linux - YouTube." [Online]. Available: https://www.youtube.com/watch?v=DWvpwSWpoiA. [Accessed: 25-Apr-2019].

[5]

"Scraping Sandbox." [Online]. Available: http://toscrape.com/. [Accessed: 24-Apr-2019].

[6]

"How to Install Docker On Ubuntu 18.04 {2019 Tutorial} | PhoenixNAP," *Knowledge Base by PhoenixNAP*, 22-Oct-2018. [Online]. Available: https://phoenixnap.com/kb/how-to-install-docker-on-ubuntu-18-04. [Accessed: 24-Apr-2019].

[7]

"¿Cómo instalar y usar Docker en Ubuntu 16.04?," *DigitalOcean*. [Online]. Available: https://www.digitalocean.com/community/tutorials/como-instalar-y-usar-docker-en-ubuntu-16-04-es. [Accessed: 23-Apr-2019].

[8]

kamilore, "Instalar Docker en Ubuntu - Tutoriales," *Tutobásico*, 06-Oct-2018. [Online]. Available: https://tutobasico.com/instalar-docker-ubuntu/. [Accessed: 23-Apr-2019].

[9]

"html - Android: how to search a word or a phrase with Jsoup," *Stack Overflow*. [Online]. Available: https://stackoverflow.com/questions/42175145/android-how-to-search-a-word-or-a-phrase-with-jsoup. [Accessed: 09-Apr-2019].

[10]

Avanzis, "Cómo Analizar a tu Competencia con Import.io," *Máster y cursos Ecommerce*. [Online]. Available: http://ecommaster.es/analisis-de-competencia-import-io. [Accessed: 08-Apr-2019].

[11]

F. Neves, "I was looking for a house, so I built a web scraper in Python!," *Towards Data Science*, 01-Nov-2018. [Online]. Available: https://towardsdatascience.com/looking-for-a-house-build-a-web-scraper-to-help-you-5ab25badc83e. [Accessed: 08-Apr-2019].

[12]

"¿Qué es el scraping de datos y cómo puedo detenerlo?," *Base del conocimiento de SiteGround*, 29-Jan-2017. [Online]. Available: https://www.siteground.es/kb/scraping-datos/. [Accessed: 08-Apr-2019].

[13]

"jsoup – Basic web crawler example – Mkyong.com." [Online]. Available: https://www.mkyong.com/java/jsoup-basic-web-crawler-example/. [Accessed: 08-Apr-2019].

[14]

"Infinispan Homepage - Infinispan." [Online]. Available: http://infinispan.org/. [Accessed: 03-Apr-2019].

[15]

J. Fox, "Regex tutorial — A quick cheatsheet by examples," *Factory Mind*, 23-Jun-2017. [Online]. Available: https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285. [Accessed: 03-Apr-2019].

[16]

"Is Python or Java considered more powerful for web scraping and automation? - Quora." [Online]. Available: https://www.quora.com/Is-Python-or-Java-considered-more-powerful-for-web-scraping-and-automation. [Accessed: 03-Apr-2019].

[17]

"How to Scrape Amazon.com Product Details and Pricing using Python | ScrapeHero." [Online]. Available: https://www.scrapehero.com/tutorial-how-to-scrape-amazon-product-details-using-python/. [Accessed: 23-Mar-2019].

[18]

"10 Herramientas de web scraping para extraer datos online," *Papeles de Inteligencia Competitiva*, 24-May-2017. [Online]. Available: https://papelesdeinteligencia.com/herramientas-de-web-scraping/. [Accessed: 23-Mar-2019].

[19]

"Example program: list links: jsoup Java HTML parser." [Online]. Available: https://jsoup.org/cookbook/extracting-data/example-list-links. [Accessed: 18-Mar-2019].

[20]

"Minería de datos," *Wikipedia, la enciclopedia libre*. 07-Mar-2019.

[21]

"Araña web," *Wikipedia, la enciclopedia libre*. 29-Nov-2018.

[22]

"Regular Expression Tutorial - Learn How to Use Regular Expressions." [Online]. Available: https://www.regular-expressions.info/tutorial.html. [Accessed: 16-Feb-2019].

[23]

"Qué es el Web scraping? Introducción y herramientas," *Sitelabs*, 08-Apr-2016. [Online]. Available: https://sitelabs.es/web-scraping-introduccion-y-herramientas/. [Accessed: 16-Feb-2019].

[24]

"Comparison of HTML parsers - Wikipedia." [Online]. Available: https://en.m.wikipedia.org/wiki/Comparison_of_HTML_parsers. [Accessed: 13-Feb-2019].

[25]

"Multitarea e Hilos en Java con ejemplos (Thread & Runnable)," *Jarroba*, 23-May-2014. [Online]. Available: https://jarroba.com/multitarea-e-hilos-en-java-con-ejemplos-thread-runnable/. [Accessed: 13-Feb-2019].

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR